DESIGN METHODS FOR SELF-REPAIRING DIGITAL LOGIC

A Dissertation
Presented to
The Academic Faculty

By

Jingchi Yang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2019

DESIGN METHODS FOR SELF-REPAIRING DIGITAL LOGIC

Approved by:

Dr. David C Keezer, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Linda S Milor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Vijay Madisetti
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sundaresan Jayaraman
School of Materials Science and Engineering
Georgia Institute of Technology

Date Approved: November 1, 2019

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. David C Keezer for the continuous support of my Ph.D. study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study. Besides my advisor, I would like to thank my fellow lab mate in research Group: Te-Hui Chen, for his guidance in Verilog programming, testing equipment, and Xilinx products. Last but not least, I would like to thank my family: my parents Jian Yang and Zhenguang Wang, for supporting me spiritually throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of this research is to establish a systematic approach for the design of self-testable, self-correcting, self-repairing and self-healing digital systems. This self-test/self-repair methodology is accomplished autonomously, in a distributed fashion, so that it can scale as the size of the system grows. Modular redundancy and re-programmability are exploited to accomplish generic self-test (applicable to nearly any application) and to enable self-repair in general, and self-healing in FPGAs. Error rates are measured throughout the design to distinguish between transient errors and permanent or semi-permanent logic faults. Original logic design is automatically transformed to this self-healing architecture via customizable software. Logic reconfiguration occurs automatically, replacing failing logic modules so that the system continues to operate error-free while partial dynamic reconfiguration is used to heal failing logic (in the case of soft faults in FPGAs).

CHAPTER 1
INTRODUCTION

The objective of this research is to establish a systematic approach for the design of self-testable, self-correcting, self-repairing and self-healing digital systems. Safety-critical hardware systems play a important role in many applications such as military systems, space applications, data centers, medical devices, and self-driving cars. These applications, where significant loss of life or destruction of capital equipment could result from failing electronics, should be designed with high reliability in mind. This research presents methods to improve digital system reliability and a technique for automatically deploying those methods.

1.1 Motivation

Computers have already become an integral part of our everyday life and most people assume that their operation will last forever and never fail. Clearly this is too optimistic. In fact, even back to the 1940s when the first generation computer is built upon vacuum tubes, errors and failures are already inevitable.

Benefiting from significant technological improvements, especially the invention of the integrated circuits in the 1960s, that fundamentally reshape the information technology industry, today's computer systems become more compact, less expensive and more powerful machines.

As Gordon Moore discussed in [1], the number of transistors in a dense integrated circuit doubles about every two years and this rate of growth would continue for decades. In fact, nowadays, even a tiny piece of silicon chip is sufficient to operate complicated task that the whole moon launch system can not achieve in the 1960s. This leads to a significant increase in the use of computers. Moreover, besides the standalone desktop

computers, at the present time, the majority of the computers are in the form of embedded systems. Although most of them are invisible to us, these devices cover a vast spectrum of applications, from sophisticated space shuttle control systems to coffee machines, and from nuclear reactor shutdown systems to automotive. The failure of many digital systems can result in direct, and possibly very serious, harm to one or more people. In some extreme instances the fault of the operation should be avoided at all costs, for example the failure of the control system of a nuclear system jeopardize millions lives. On a smaller scale, failures in an automobile brake system could potentially kill dozens of people. Even if no human lives are under threat, some failures in digital systems could lead to catastrophic financial loss.

Moreover, reliability in hardware systems is facing the challenge of the increasing complexities of modern digital systems as well as the decreasing dimension of semiconductor features. As a result, in the near future, even non-safety-critical designs might need to utilize safety-critical techniques in order to meet minimum reliability requirements.

As can be seen from the above discussion, the operation of many digital systems has a direct influence on the user safety and user property. And to achieve such a highly reliable and self-repairing electronic system, engineers must acquire expertise not only in domain-specific application design but also safety-critical design. Alternatively, we present a wrapper system that automatically upgrades non-safety-critical design into a more reliable version according to the specific system-level requirements, such as self-testing, fault tolerant and self-healing for Field Programmable Gate Array (FPGA) based design.

1.2 Approach

Biological systems have evolved through billions of years and thrive largely due to their ability to tolerate, repair, and heal defects. Most of the energy expended in living systems is devoted to their top priority, which is to stay alive. In our work, we follow nature's example and put our priority on survival mechanisms in electronics, starting with self-test,

repair, and healing.

At a low-level, the DNA code present in all cells is made up of only four types of small molecules arranged as complimentary base-pairs. This low-level complimentary paring represents a first level of redundancy, somewhat analogous to the complementary operation of CMOS logic. Defects in a base-pair are recognized by repair enzymes that sense that a base molecule is either missing or miss-paired, and initiate repair mechanisms to replace the defect with its proper base molecule.

Self-test is enabled within biological systems by the complementary pairing of these bases. It provides a natural way to identify single-point defects. In this sense, the double-helix duplication of the DNA code is analogous to dual-modular redundancy (DMR) in electronic systems. It provides an automated way of detecting faults and serves as the starting point for repair. At an even higher level, nature duplicates entire organs within the body. In most cases the individual can survive (although in a degraded condition) with the loss of one of the duplicate organs.

In this research, self-testing is achieved through an enhanced modular redundancy technique that not only compares two or more identical modules, but also distinguishes intermittent errors from temporary transient errors. This distinction is critical for the fault repairing process since transient errors usually only last for one clock cycle, thus by the time the repairing process starts the error might already disappear. Therefore, correcting the corrupted data caused by the transient errors is recommended and the traditional error detection method, which flags every error, is not efficient for identifying modules for repair. By filtering out transient errors from intermittent errors, the repairing procedure required by the fault management unit is effectively reduced by 90% to 95%, which significantly reduced the average single module offline time, and thus increases the system availability overall. Moreover, the hardware implementation cost of this enhanced logic is negligible. Therefore, this minimal transistors or gates requirement can largely increase the enhanced module scalability.

To achieve fault tolerance, we introduce an enhanced Triple Modular Redundancy (TMR) technique that automatically corrects transient errors and identifies permanent error for requesting repairing procedure more efficiently. To maintain the fault tolerance during fault repair process and to tolerant more than one irreversible hardware defects, an enhanced Quadruple-Modular Redundancy (QMR) technique is presented. On the one hand, combining all these techniques, a self-repairing architecture that satisfies high reliability requirement is achieved. On the other hand, depending on the reliability requirement, the individual design technique can be selected or composed to achieve desired safety performance.

This self-repairing architecture is easy to scale and can be applied on different hierarchy of the circuit. Furthermore, this highly reliable system architecture can be implemented in both application-specific integrated circuit (ASIC) or Field Programmable Gate Array (FPGA). However, if this system is implemented in FPGA, it can also benefit from a healing process based on the FPGA partial reconfiguration technique. A healing controller that manages this reconfiguration process is also introduced.

In addition to this self-repairing architecture, the deployment of such an architecture should require minimal extra effort. Therefore, the transformation from a non-safety-critical design to a reliable version is accomplished automatically with a software program. This is extremely useful for converting extremely large design that contains of thousands of modules. This method also eliminates human interaction, therefore increases the productivity while avoiding the human error.

Combining all these methods and techniques, a framework for the design of self-testable, self-correcting, self-repairing and self-healing digital systems is introduced. As presented in Figure 1.1, on the software side, this framework contains an HDL converter for automatically modifying the original Verilog source code. From a hardware point of view, this framework utilizes FPGA and an external healing controller for deploying the self-repairing architecture and managing the healing procedure. The general steps of prototyping a self-

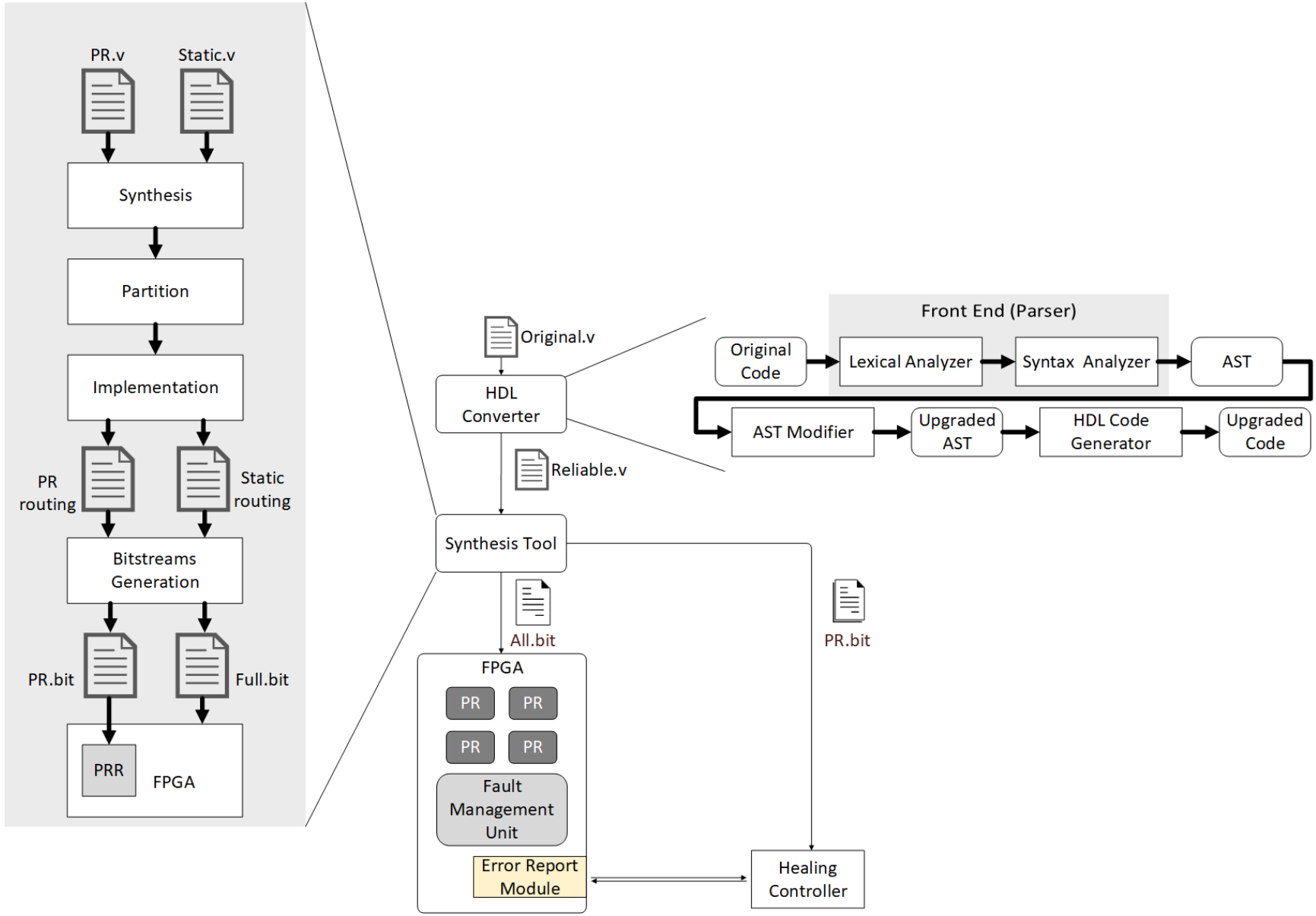

Figure 1.1: Development work flow.

repairing hardware system are discussed as follows:

First, the HDL converter reads the original HDL file and modifies it to deploy the state synchronization technique. Then it creates a higher level module with four redundant user logic instances, one enhancement module and one interface module.

Second, the synthesis and implementation software generates a main bit-stream file and several partial reconfiguration bit-stream files from those HDL files. The main bit-stream is used to program the FPGA and the partial reconfiguration bit files are used as needed for healing the damaged modules.

Finally, when an error is reported, the healing controller will reprogram the faulty area with the corresponding partial bit stream.

To summarize, this research provides an universal solution to the design of highly reliable digital system. The conversion from a non-safety critical design to self-repairable and fault tolerant design is achieved automatically. This updated design is autonomous and can be easily scaled up. Various digital designs of different scales are implemented and examined under random error injection. The test results demonstrate the effectiveness of this design.

The methods and techniques presented in this thesis offer six distinct contributions as presented below:

- Bit Error Rate (BER) Measurement

  A bit error rate enhanced testing logic filters out transient errors from soft errors, leading to a 90% to 95% reduction in the repairing request.

- Enhancement for Fault Tolerance and Fault Isolation

  An enhanced voting logic not only corrects the transient error generated by the faulty module, but also automatically identifies the failing unit for the repairing procedure.

- Self-Repairing Architecture

  An architecture with the ability of self-testing, fault-tolerance and immediate context switching when a permanent fault are detected. It is easy to scale and can be applied on different hierarchy of the circuit.

- State Synchronization

  A method to synchronize the sequential state of the newly repaired module with the nearby health module. Without this technique, the entire system must be re-initialized in order to reuse the repaired module.

- HDL converter

  A Verilog compiler based software that automatically modifies the user logic and upgrades it into a more reliable version. This method eliminates the human interaction, therefore increases the productivity while avoiding the human error.

- Self-Repairing Design Framework

  A framework for quickly prototyping a self-testable, self-correcting, self-repairing and self-healing digital systems.

1.3 Thesis Organization

This thesis is organized as follows: In Chapter 2 “Background and History”, self-testing and self-repair techniques are introduced. In Chapter 3 “Self-repairing Methodologies”, details about the self-repairing system architecture is presented. In addition, a method of synchronizing the state register between different modules is introduced. In Chapter 4 “Healing Controller”, details of using dynamic partial reconfiguration to repair failed logic blocks in FPGA application are revealed.

The remaining research presented in this thesis uses the methods and techniques discussed in Chapter 3 to automatically upgrade non-safety-critical design into a more reli-

able version. In Chapter 5 “Software Toolchain”, mechanism of autonomously applying the self-repairing architecture to any given digital design is explained. In Chapter 6, “Experiment results” are presented to demonstrate the reliability improvement and the costs of various example design ranging from SSI (small scale integration) to VLSI (very large scale integration). In Chapter 7, a brief summary of the thesis work is presented. In addition, the achievement of this research is highlighted, followed by the conclusion and future works.

CHAPTER 2
BACKGROUND AND HISTORY

No hardware system is entirely error-free [2]. This is first due to the imperfection of the system design, and secondly the hardware component failures caused by wear, ageing or other effects such as noisy power supply, high environment temperature or an alpha particle [3].

For the past few decades, varies methods and techniques aimed for the digital system safety requirement have been developed under the assumption that design and component faults are inevitable. These measures can be categorized into three groups of techniques:

- fault detection
- fault repair
- fault tolerance

Approaches to eliminate faults in digital system are related to the cause of those faults. Therefore, it is relevant to discuss the origin of those faults to fully understand the logic behind the safety-critical design techniques.

Due to this reason, the cause of faults in digital systems are discussed first in this chapter. To achieve a high reliable system, we must be able to identify the fault. Therefore, after presenting the source of errors in digital system, a brief review of studies in fault detection is elaborated. After the fault is detected, we need to repair it to stop the negative effect. Thus the discussion about the fault repair techniques is presented.

In consideration of the down time caused by the repairing process, systems with high availability requirement should also be able to produce the correct result despite the existence of errors. This leads to the concept of the fault-tolerant system [4] and is exhibited in the last section of this chapter.

2.1 Faults and the Cause

Different reference sources categorize faults based on different standards. For example, faults can be divided into two separate classes based on their locations [3]. The first of these is random faults. As the name suggests, these type of faults can occur anywhere (and also anytime) in the circuit. They are primarily related to the individual hardware component failures. The second type is termed systematic faults. These faults usually stay in one area of the system since they are mainly due to the imperfect design.

Another commonly viewed categorization is based on the fault duration [5]. They are permanent faults and transient faults. Sometimes intermittent faults are also included.

2.1.1 Permanent Faults

Permanent faults are typically caused by an irreversible physical change. In fact, the occurrence of permanent fault is directly associated with the semiconductor design and the manufacturing techniques [3]. For convenience, we use permanent fault and hard fault interchangeably in later discussion.

2.1.2 Transient Faults

Transient faults are faults caused by temporary anomaly on the hardware device. There are several non standard conditions induce transient faults [5]:

- power supply noise
- neutron and alpha particles

Fault Source 1: Power Supply Noise

In large circuits, power is distributed in the circuit through wires, which contain parasitic devices. The power supply noise is mainly caused by resistive and inductive parasitic

elements in the power supply lines. The relationship between the power supply noise and the parasitic elements is illustrated in the following equation [6]:

$$\Delta V = IR + L\frac{dI}{dt}$$

where R and L are the wire resistance and inductance respectively. When the logic gates are switching, the current which flows through the power supply lines will cause the power supply voltage to drop. This voltage drop increases the gate sensitivity to noise spike, which consequently increases the circuit error rate [7].

Fault Source 2: Radiation

The cause of radiation-induced transient faults, e.g. when the neutron or alpha particles strike the semiconductor chip, is related to the metal-oxide-semiconductor field-effect transistor (MOSFET) working principle [8]. High energy electrons such as alpha particles can ionize the atoms and generate enormous electron-hole pairs [8]. As illustrated in Figure 2.1 and Figure 2.2, (a) High energy particles penetrate into the semiconductor substrate while producing enormous electron-hole pairs along the path. (b) Those generated electrons close to the depletion region are attracted by the depletion region electric field; similarly the generated holes are driven away from the depletion region. This behavior further increases the size of the depletion region and force more electrons and holes to drift in the opposite direction, which creates a large spike of current. (c) Similar to the mechanism of a PN junction, the difference in electrons and holes concentration leads to a diffusion into the opposite side, which counteracts the previous movement, re-balances the electrons/holes concentration and returns the depletion region to its original size. If the current pulse or the charges accumulated during this event reaches certain threshold, it will generate an error. However the circuit itself is not damaged [9].

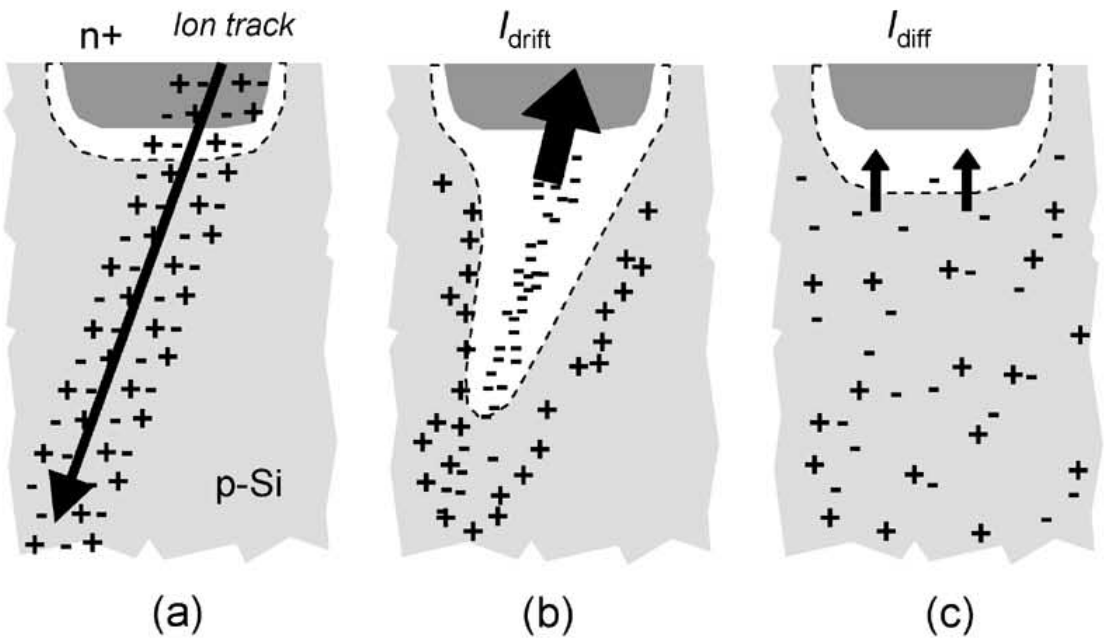

Figure 2.1: Charge generation and collection phases in a reverse-biased junction [8]

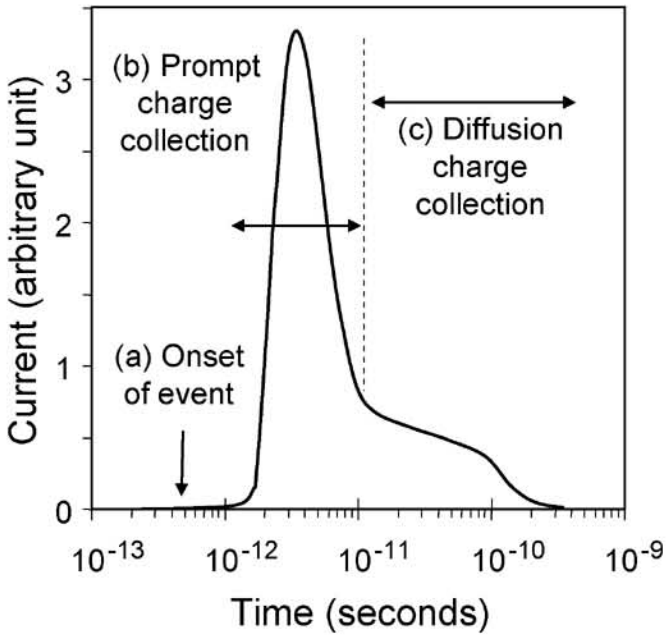

Figure 2.2: Current pulse caused by the high-energy ion [8]

Similarly, collisions induced by high energy particles can also occur in the oxide layer. As a result, charges build up in the oxide can also cause device failure. The current leakage during the switch "Off" state is induced by the positive charge trapping in the gate oxide, which increases the static power consumption and may also cause transient failure. Conversely, positive charge build-up in the field oxide can also reduce the current when the switch is at "ON" state. In fact, for modern integrated circuits with ultra-thin gate oxides, the majority of the radiation-induced degradation is caused by charges build-up in field oxides [10].

Neutron impacts are similar except that they do not generate the electron-hole pairs directly. Instead, collisions cause by neutrons striking the silicon crystal produce alpha and other high-energy particles. These particles can generate sufficient electron-hole pairs to produce a transistor error.

Even though alpha particles and neutrons are both classified as radiation, they derive from different sources. Alpha particles arise from chip packaging material while the dominant source of neutrons comes from the cosmic rays [11].

2.1.3 Intermittent Faults

Intermittent faults are similar to transient faults, therefore it is difficult to distinguish one from the other. Constantinescu and Cristian provide three main principles to judge the class of fault that caused an error [5]. First, an intermittent fault tends to be more static than transient errors, which means it occurs repeatedly at the same location. Second, intermittent errors are more likely to occur in bursts when the fault is activated. Third, replacing the faulty logic removes the intermittent fault, while transient errors can not be eliminated in this way. In Chapter 4 Self-repairing Methodologies of this thesis, we provide a method to distinguish intermittent faults from transient errors. For convenience, we use intermittent and soft error interchangeably in later discussion.

2.2 Fault Detection Methodologies

Fault detection methodologies can be categorized into two approaches: on-line and off-line [12]. On-line testing is executed during system run time while the off-line approach is executed when the system is not operating. Since off-line test results do not guarantee that on-line operation will be error-free, the discussion here will be restricted to on-line testing techniques. On-line testing schemes usually depend on redundancy techniques. For hardware error detection, there are three forms of redundancy techniques: hardware, information and time [2][12].

2.2.1 Hardware redundancy

Hardware redundancy schemes utilize extra hardware to generate a reference module and compare the reference results with the module under test. The easiest way to implement a reference hardware module is to duplicate it. This method is also known as Dual Module Redundancy (DMR). As illustrated in Figure 2.3, by executing the same task on two identical modules and comparing their outputs, as long as both modules do not make the same mistake at the same time, a DMR system can detect any error whenever a disagreement in the outputs occur.

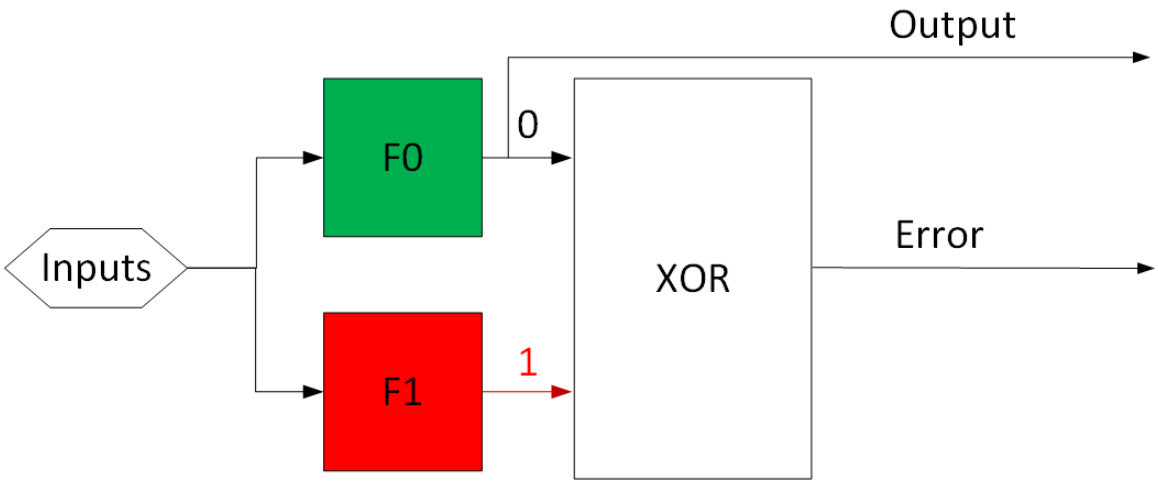

Figure 2.3: Dual module redundancy diagram.

Besides the classic dual modular redundancy (DMR) test methodology, Burress et al propose a look-up table (LUT) based logic cell that operates on the premise of the two-rail checker. Firstly, according to the sum of products, the function will be realized by a series of product and sum logic blocks. Each block has two outputs and no more than four inputs. The block is designed in such a way that it can generate complementary outputs when the cell works well and identical outputs in the event of faults [13]. A script in [14] is used for simplifying and decomposing the Boolean expression. This methodology requires less area overhead (78%) compared with the classical DMR technique. However, any modification in the redundant module may vary the data path delay, and thus violate the timing requirement of the original module. Therefore, this approach requires extra development time and effort to solve the potential synchronization issue.

2.2.2 Information redundancy

Information redundancy techniques utilize extra check bits to test the original data bits [15][16]. This is also known as the error detection coding (EDC) which includes parity code, m-of-n code and Borden's code [15]. Instead of checking every single bit from the output, only the encoded check bits are compared to detect the error. Although these methods are mostly used in data communication, studies have applied these approaches in modular function testing. For example parity check has been used in adder and arithmetic and logic unit (ALU) error detection by implementing a parity predictor [16].

A parity predictor is an optimized implementation of the original module plus the parity bit encoder [17]. Since the number of parity bits is generally smaller than data bits, parity prediction usually requires less area overhead compared with the DMR system [16].

The optimization of the implementation of parity prediction in ASIC has been studied for years. Sobeeh et al present a parity-tree-selection method based on the relation between the entropy of a function and the area and power-consumption overhead incurred by its implementation [18]. The idea is to use entropy theory to find the optimal number of

parity bits. The author compares the entropy-driven methodology and the minimum parity bits strategy and proves that the entropy driven parity-tree-selection method costs less area overhead. On average this technique costs 60% overhead. However, it should be noticed that in some cases, the parity check system cost even more area than DMR.

Mitra et al summarize three different concurrent error detection (CED) schemes [19]: duplex system, parity prediction and unidirectional error detecting code [15]. The conclusion is that the parity prediction technique has the lowest area overhead while the diverse duplex technique is the most reliable approach. Although error detection coding has an advantage in hardware overhead, some techniques are stricted to specific types of errors [15]. Moreover, similar to the diverse DMR approach, synchronization with the unit under test is a potential issue.

2.2.3 FPGA test methods

VLSI testability has been summarized to controllability and observability, yet FPGAs have unique properties, and thus meet a new challenge in testing [20]. Due to the fact that area consumption in FPGAs is related to the number of inputs of a function rather than the complexity of the function itself [21], Seok et al propose a new method for implementing the parity prediction function in FPGAs [22]. The idea is to decompose the original parity function into several small-input-number functions in order to fit in with the standard lookup table (LUT). However, all parity prediction methods have the same synchronization issue as discussed previously.

Mehdi et al present a technique for FPGA testing by reconfiguring the interconnection routing logic and the configurable logic block (CLB) [23]. When testing the routing logic, the configuration of the interconnection routing logic stays unchanged. But the CLBs will be reconfigured to implement a single-term function in order to test the interconnections and vice versa.

Miron et al present an FPGA test strategy called Roving STARs [24]. In order to test the chip without interrupting the functionality, the chip is divided into many blocks. Only a small part of the chip will be tested each time while the other parts keep functioning. Once the test is completed, the test target moves to the next area and repeat the previous process. The circuit blocks in the test zone are switched between three roles: test pattern generator (TPG), output response analyzer (ORA) and Block under test (BUT). To be more precise, two identically block are tested by a test pattern generator (TPG) and those outputs are then sent to an output response analyzer (ORA). The system was designed in such a way that each block will be tested twice and each time compared to a different block. Eventually, all the chip will be tested.

Nathan has a similar technique for bus-based FPGA [13]. The drawback of the above methods is that due to the configuration time cost, the speed of switching testing area is very slow. Therefore, the developer has to make a trade-off between the operating clock frequency and the system availability.

2.3 Fault Repair Methodologies

Although the above strategies can detect errors, they cannot repair errors. Thanks to the flexibility of FPGA, single event upset (SEU) in configuration memory can be repaired by reprogramming the configuration bits [25]. However, permanent physical damage in FPGA cannot be repaired, but the faulty module can be replaced with the redundant module.

For example, Herrera-Alzu et al present a background configuration memory refresh technique which allows the FPGA to check and correct the configuration error without interrupting the application operation [26]. Stoddard et al introduce a hybrid configuration scrubbing approach with an external controller [27]. M Berg et al compared two configuration scrubbing techniques and present the experimental results under a radiation test [28].

Dumitriu et al propose a novel architecture for transient and hardware faults online recovery [29]. Transient errors or SEUs can be repaired by reconfiguring the faulty area

of the device. Logic circuitry containing hardware faults will be reallocated to spare area through dynamic partial reconfiguration. The hardware rerouting process is replaced by changing the broadcast mode in the large central bus interface in order to isolate the faulty module and connect the relocated module to the system. Although the area overhead seems little at first glance, the cost of the central bus is actually significant, nearly 50% of the total area. Moreover, due to the high area cost, such a safety-critical central bus is more vulnerable to defects compared with the subsystem module.

Kim et al. propose a novel self-repairing architecture inspired by paralogous genes [30]. This design has a built-in self-test module. Once a fault is detected in the working cell, the input will be redirected to a redundant cell with the same functionality. The faulty cell will be self-tested. For transit errors, the faulty cell will become a redundant cell after self-test. In the case of permanent error, the faulty cell will be programmed to be a "death cell" and a "stem cell" will be programmed as a new redundant cell through partial reconfiguration.

However, these self-repair methods all face two main problems. First, the speed of the configuration memory scanning is extremely slow compared with the clock frequency. Meanwhile, errors could propagate to other units long before being detected. Second, the repairing process is not efficient when the transient error rate is significantly higher than other permanent errors. Since the transient errors last briefly, by the time the repairing process is finished, they are already disappeared.

2.4 Fault-Tolerant Techniques

Similar to the error detection techniques, fault-tolerant designs also depend on redundancy techniques. In general, hardware redundancy schemes cost more area overhead while information and time redundancy schemes have a larger performance penalty [2]. Since in most cases, logic functions that implemented in FPGA do not use the entire logic resource [31], small designs could take advantages of the unutilized area and implement the redundant modules with those resources. In addition, the cost of the hardware resources is expected

to keep decreasing [32], thus the disadvantage of the hardware redundancy becomes less important in the long-term. Therefore, the discussion here focuses on hardware redundancy techniques over information and time redundancy for fault-tolerant design methodology.

Triple Modular Redundancy (TMR) is one of the most widely used fault-tolerant strategies [33]. As presented in Figure 2.4, by tripling the original design and voting the result with a majority gate, as long as two out of three modules work correctly, errors generated by the faulty module will be masked by the majority vote. Xilinx provided a method for the TMR design in their Virtex FPGAs [34]. These approaches are effective against transient errors however they ignore the consequence when one module continuously produces errors [35].

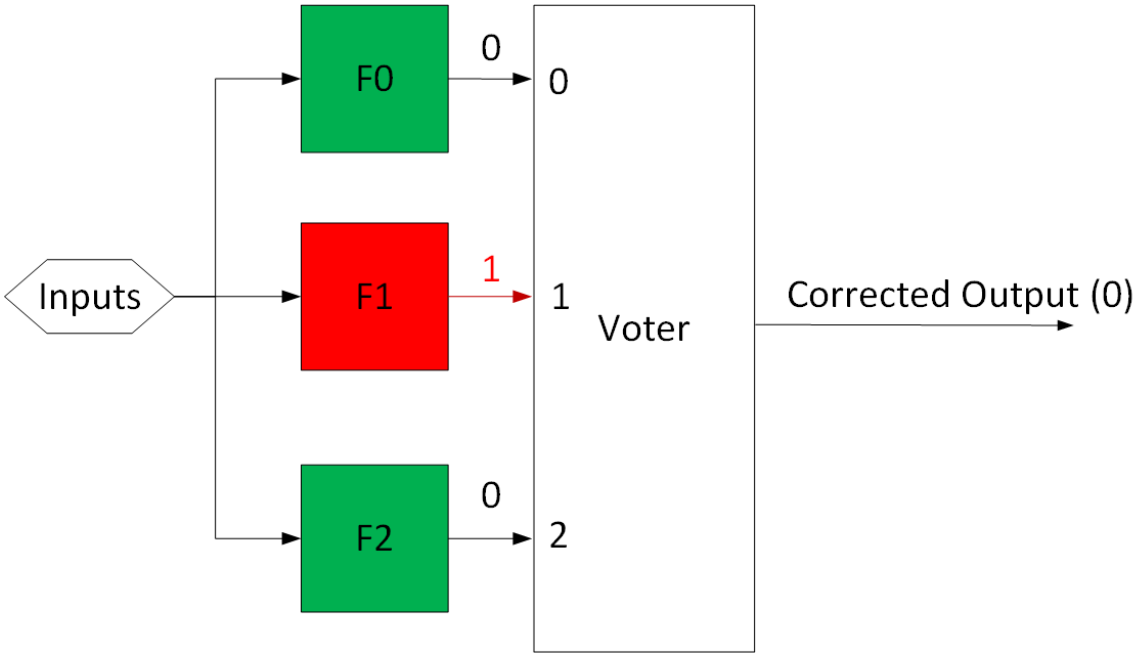

Figure 2.4: Traditional TMR diagram.

Mathur et al proposed a TMR system with self-repair techniques integrated [36]. This system replaces the faulty module with a spare module. However, without the distinction between a transient error and permanent error, spare modules will soon be exhausted.

Zhang et al developed an algorithm to generate a minimal set of configurations for

one functional module, each configuration in that set has at least one spare CLB [37]. This minimal set covers all possible locations for the single CLB-faults and thus achieves single CLB-faults tolerant. Moreover, alternating the spare CLB with the heavy-duty CLB balances the system stress thus increases the overall system lifetime.

Baig et al introduce a hierarchical fault-tolerant architecture [38]. At the top level, the system contains multiple computation tiles, each tile contains multiple computation blocks. Inside the block, there are various computational cells and preserved spare logic resource called stem cell. Each computational cell is designed to be fault-tolerant with pre-computed error detection code and spare look-up tables. Those spare resource, either stem cell or spare look-up table is designed for function relocation at a different level in case of permanent error occurs. However, these methods cannot restore the sequential logic state in the faulty logic, the data stored in the flip-flop will be lost when switching to a spare logic.

DeMara et al. proposed a TMR based fault-tolerant architecture for image processing application [39]. In order to minimize the power consumption, the third module will not be active until the discrepancy is detected by DMR. The faulty module is repaired through intrinsic Genetic Algorithm.

Oreifej et al. also provide an example of using GA for reconfiguration [40]. An enhanced version of GA called Combinatorial Group Testing (CGT) is proposed. Refurbishing partially-functional configurations are demonstrated to be more efficient than designing the configurations when using genetic algorithms. However, even with the advanced algorithm, the the process of generating a valid configuration bitstream from the individual arbitray attempts is still extremely time-consuming.

CHAPTER 3

SELF-REPAIRING METHODOLOGIES

As introduced in chapter 2, various self-testing and self-repair strategies have been utilized in the past. However, they all have certain limitations. Our proposed methodology recognizes that the optimal level of fault-tolerance and repairability will be application-specific. Therefore we propose the following strategies that can be deployed, depending upon system fault-tolerance requirements and the Mean Time Between Failure (MTBF) of the components (logic blocks). Enhanced-DMR can distinguish different kinds of errors but not correct them. Enhanced-TMR can correct transient errors but not repair faulty modules with permanent faults. To overcome these limitations, we develop the enhanced-QMR technique which can autonomously replace the soft/hard-failing unit in order to maintain TMR operation. These strategies not only can be applied independently to the existing error detection and error correction technologies but also can be integrated together to achieve very high reliability. Combined with the autonomous partial reconfiguration technique [41], FPGA applications can extend the system lifetime by centuries [42]. The enhanced-QMR approach is particularly applicable to extremely large-scale systems where errors are expected to be frequent.

3.1 Enhanced-DMR (with BER-measurement and error-reporting)

As presented in the previous chapter, transient errors by their nature appear randomly and sporadically [2]. Moreover, transient errors in configuration memory do not necessarily lead to a soft functional error. The erroneous bit has to be one that is critical to the function in order for a soft functional error to be observed. The number of unused bits and non-critical errors reduces the typical soft error rate to 5% to 10% [43] (only one in 10 to 20 upsets, on average, cause a functional soft error). Therefore the traditional DMR method,

which flags every error, is not efficient. A better solution would be to detect functional errors but ignore or correct non-recurring transient errors so that higher-level repair mechanisms can be invoked at the system level only when there is a high likelihood of a structural fault.

Beyond sporadic transient errors, soft or hard errors occur when a fault causes a change in the logic behavior. In FPGAs, most soft errors result from faults in the configuration memory, which can be corrected by reprogramming [25].

In order to distinguish the transient errors and the soft errors, a bit error rate measurement logic is introduced. In Figure 3.1, each error is not only recognized, but is counted, and compared to a threshold BER to distinguish between common transient errors (isolated, single-bit errors) and soft/hard errors (resulting from changes in logic structure).

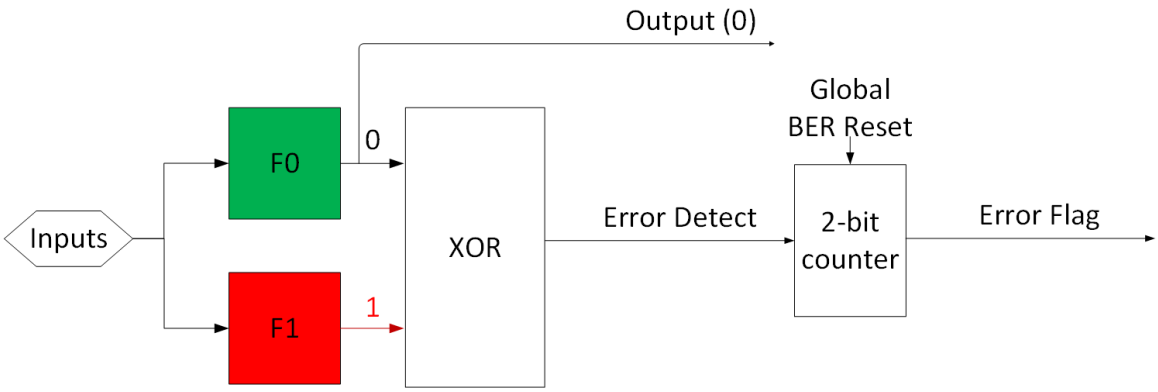

Figure 3.1: BER enhanced-DMR diagram.

In an FPGA, a change in the configuration memory will often cause a permanent change in the logic behavior, resulting in a significant increase in error rate. A local 2-bit counter is used to count the errors between global resets and stopped counting at a count of 2 (or optionally 3). The reset signal is produced by a shared global counter and resets the local 2-bit counters periodically. This periodic global reset signal is designed to eliminate the potential false alarm caused by the accumulation of transient errors during a long period operation.The number of bits in the global counter is chosen based upon the expected ac-

ceptable BER and an unacceptable rate that is twice as high. As illustrated in Figure 3.2, the 2-bit counter is designed to output only when the higher value is reached, signaling the system using the Master Error Flag signal. Therefore, random transient errors can be distinguished from soft errors which cause faults at much higher rates. As can be seen in the later section, this BER enhancement can also be applied to higher level methods.

In conclusion, the enhanced-DMR features and benefits are presented as follows:

- Provides basic self-test capability, repair requires system-level operations
- Most soft faults can be corrected by re-programming the failed functional cell and tester
- Distinguishes transients from soft defects
- Low overhead cost (compared to TMR, QMR)

On the contrary, the enhanced-DMR limitations are summarized as follows:

- Assumes very low probability of errors
- Does NOT correct transient errors
- Normally simple-DMR does not provide fault-isolation or repairability
- Normally simple-DMR does not distinguish between transient, soft, or hard errors
- Assumes that the system can tolerate the time needed for repair of soft-defects (by re-programming the failed cell)
- If time-multiplexing is not used, then the entire cell must be reprogrammed and it will be temporarily non-functional during the reprogramming step
- Suitable only for non-critical applications
- Best suited for homogeneous systems and systems that can tolerate temporarily-failing logic

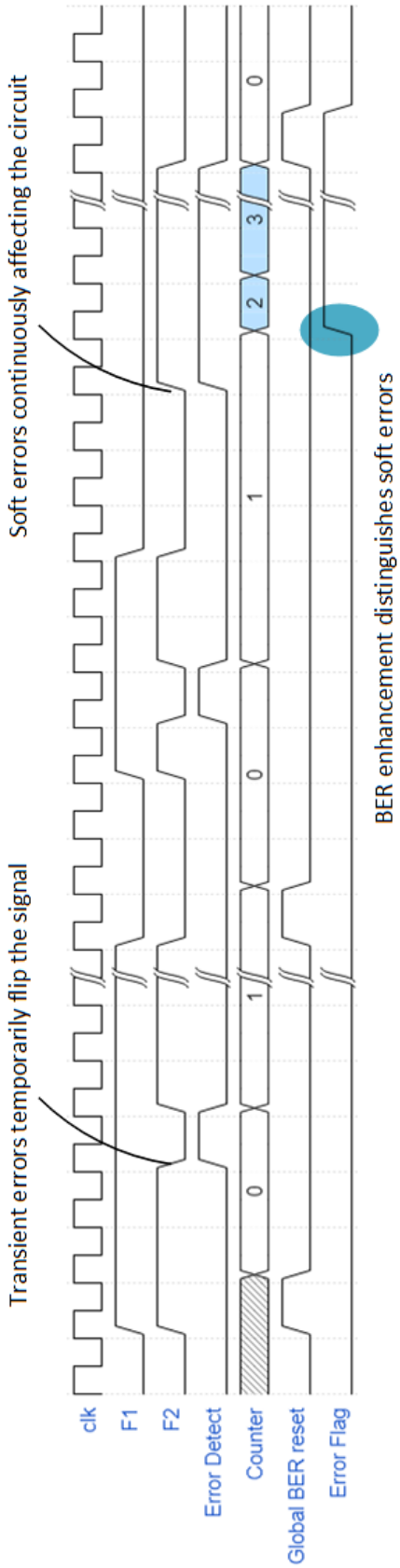

Figure 3.2: BER enhanced-DMR timing diagram.

3.2 Enhanced-TMR

Although the enhanced-DMR approach can identify critical intermittent errors, it cannot correct these or transient errors. Even though the bit error rate (BER) for transient errors is low, for example, the BER for Xilinx 7 series FPGA 7k325T is 5.69×10^{-15} [30], it is still unacceptable for safety critical applications. Moreover, certain circumstances such as radiation will tremendously increase the BER [44] and cause SEU in configuration memory [45][46]. Therefore, for applications with higher level reliability requirements, an error recovery or fault tolerant mechanism is necessary. The traditional TMR method is capable of masking one error as long as the other two outputs are error free. However, studies have proven that once a module in the TMR system is failed, the reliability of such a system is inferior to that of a single module system [35]. Therefore, in order to maintain the benefit of a highly reliable TMR based fault tolerant system, a fault identification and self-repairing mechanism is required.

In our enhanced-TMR approach, fault-isolation for repair of intermittent defects is added to the traditional TMR. Like our enhancement to DMR, we add the ability to set a BER threshold, above which the unit is suspected of having a soft or even a hard fault. As shown in Figure 3.3, we use three copies of the logic block (like traditional TMR) and enhanced voter logic to not only carry out the voting, but also to count the BER and identify which of the three blocks has failed. A 2-bit error code is generated that identifies the most recent failing block, as presented in Table. 3.1. For example, on the second row, when F0 and F1 produce a logic "1" while the F2 module generates a logic "0", the enhanced voter identifies the disagreement between the three modules and activates the error flag.

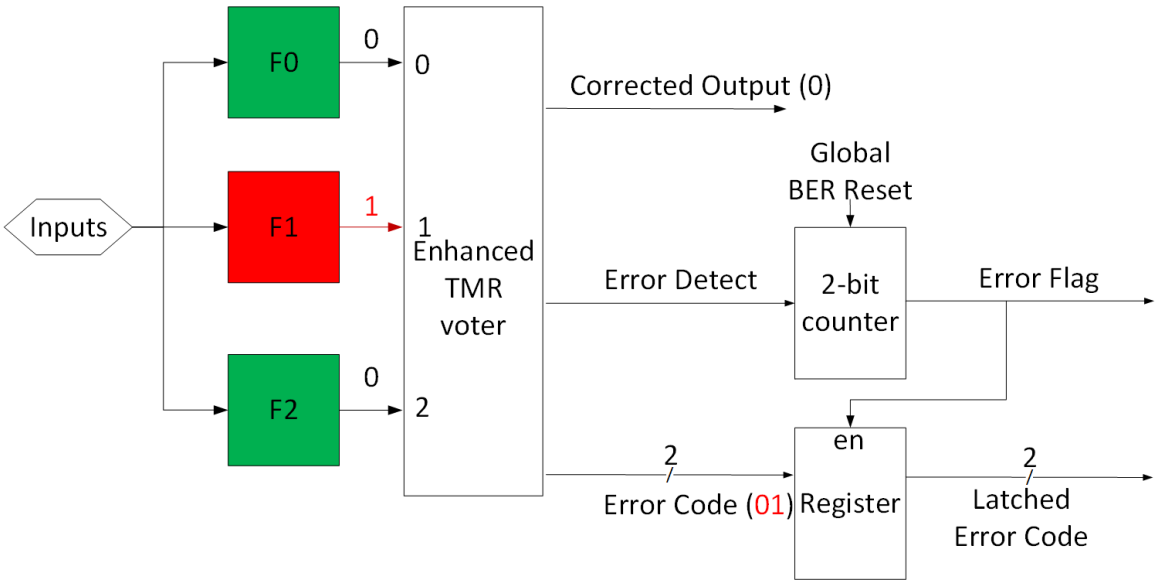

Figure 3.3: Enhanced-TMR with Voter and BER logic.

Meanwhile, it also generates the error code "10", indicating the error is produced by F2. If no error is caught, as shown in the first row and last row, then the error code is not valid. The "xx" in the table represents "don't care value". This error code is used by the higher-level system maintenance hardware to determine which block to attempt to repair (by reprogramming). The higher-level system maintenance hardware could be implemented using

Table 3.1: Truth Table of enhance voter logic with TMR

F0	**F1**	**F2**	**out**	**err_code**	**error_flag**
0	0	0	0	xx	0
0	0	1	0	10	1
0	1	0	0	01	1
0	1	1	1	00	1
1	0	0	0	00	1
1	0	1	1	01	1
1	1	0	1	10	1
1	1	1	1	xx	0

the autonomous partial reconfiguration technique [41]. During the repair attempt, we allow the two remaining non-failing units to continue to function (in DMR mode). While in

DMR mode, the frequent transients generated by the failed/re-programming block are ignored by the voter logic, so the system is temporarily reduced to DMR mode while filtering out ignoring the transients from the failed unit.

In summary, the enhanced-TMR features and benefits are presented as follows:

- Automatically corrects the vast majority of transient faults (like normal TMR)
- Uses BER to distinguish transients from soft defects (enhancement)
- Automatically identifies the failing unit and reports it for eventual repair (enhancement)

On the other hand, the enhanced-TMR limitations are shown as follows:

- Temporarily drops into DMR mode while a soft defect is repaired/healed
- Repair/healing requires assistance from higher levels of the system
- Overhead is 200% (plus the cost of BER and error-reporting logic)

3.3 Enhanced-QMR (TMR + local spare)

The enhanced-TMR approach described above exhibits degraded fault-tolerance and BER (effectively DMR) between the onset of failure and completion of the reprogramming step. This degradation may not be tolerable by some very critical systems. Worse, if a hard error is encountered (not repairable by reprogramming the failing logic block), then the system has no convenient way to recover without globally reconfiguring (rerouting) a major portion of the system. Even if this is possible, it will result in a long down-time.

In our enhanced-QMR approach, we add a fourth logic block to the enhanced-TMR approach and modify the voter/BER/Error-code logic to account for the fourth block. This scheme is shown in Figure 3.4. Initially, the voter logic produces a 2-bit internal ignore code that effectively masks one of the four logic blocks outputs (the backup spare). It then

treats the remaining three logic blocks like a TMR configuration. As in our enhanced-TMR scheme (above), transient errors are counted and compared to a threshold BER.

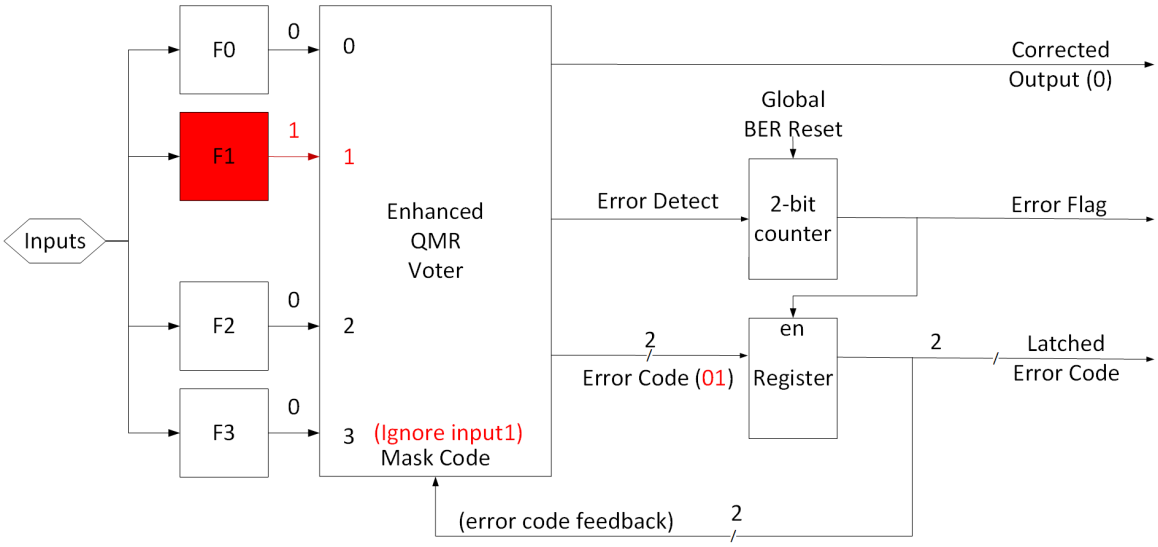

Figure 3.4: Enhanced-QMR with Voter and BER logic.

If the BER remains below a predetermined threshold, then the spare unit is ignored and the system proceeds to correct transients in TMR fashion. If the BER exceeds the threshold, then the last failing bit identifies the suspected failing unit. That 2-bit code is latched and internally fed-back to signal the voter to ignore the suspected failed unit while activating the spare (fourth) unit, thereby seamlessly replacing the failed unit and bringing it offline for reprogramming. Table. 3.2, Table. 3.3 and Table. **??** present the enahnced-QMR voter logic truth table.

This ability of maintaining TMR with no down time guarantees that no transient error will pass to the output. Moreover, this repair mechanism does not require external control and consequently provides good scalability. However, external control may be used to attempt healing of the failed block by reprogramming the configuration memory. For example, recent research has demonstrated the FPGA self-healing process using the autonomous partial reconfiguration technique [41]. It is capable of detecting and correcting upsets in configuration memory by scanning and correcting the configuration frame. After

reprogramming, the system is reset back to the original configuration and the reconfigured unit is reactivated in enhanced-TMR fashion (the fourth unit is ignored).

Table 3.2: Fault Management Unit state table 1.

err_code (input)	**F0**	**F1**	**F2**	**F3**	**out**	**err_code (output)**	**error_flag**
11	0	0	0	0	0	xx	0
11	0	0	0	1	0	xx	0
11	0	0	1	0	0	10	1
11	0	0	1	1	0	10	1
11	0	1	0	0	0	01	1
11	0	1	0	1	0	01	1
11	0	1	1	0	1	00	1
11	0	1	1	1	1	00	1
11	1	0	0	0	0	00	1
11	1	0	0	1	0	00	1
11	1	0	1	0	1	01	1
11	1	0	1	1	1	01	1
11	1	1	0	0	1	10	1
11	1	1	0	1	1	10	1
11	1	1	1	0	1	xx	0
11	1	1	1	1	1	xx	0
00	0	0	0	0	0	xx	0
00	0	0	0	1	0	11	1
00	0	0	1	0	0	10	1
00	0	0	1	1	1	01	1
00	0	1	0	0	0	01	1
00	0	1	0	1	1	10	1
00	0	1	1	0	1	11	1
00	0	1	1	1	1	xx	0
00	1	0	0	0	0	xx	0
00	1	0	0	1	0	11	1
00	1	0	1	0	0	10	1
00	1	0	1	1	1	01	1
00	1	1	0	0	0	01	1

Table 3.3: Fault Management Unit state table 2 (continued).

err_code (input)	**F0**	**F1**	**F2**	**F3**	**out**	**err_code (output)**	**error_flag**
00	1	1	0	1	1	10	1
00	1	1	1	0	1	11	1
00	1	1	1	1	1	xx	0
01	0	0	0	0	0	xx	0
01	0	0	0	1	0	11	1
01	0	0	1	0	0	10	1
01	0	0	1	1	1	00	1
01	0	1	0	0	0	xx	0
01	0	1	0	1	0	11	1
01	0	1	1	0	0	10	1
01	0	1	1	1	1	00	1
01	1	0	0	0	0	00	1
01	1	0	0	1	1	10	1
01	1	0	1	0	1	11	1
01	1	0	1	1	1	xx	0
01	1	1	0	0	0	00	1
01	1	1	0	1	1	10	1
01	1	1	1	0	1	11	1
01	1	1	1	1	1	xx	0
10	0	0	0	0	0	xx	0
10	0	0	0	1	0	11	1
10	0	0	1	0	0	xx	0
10	0	0	1	1	0	11	1
10	0	1	0	0	0	01	1
10	0	1	0	1	1	00	1
10	0	1	1	0	0	01	1
10	0	1	1	1	1	00	1
10	1	0	0	0	0	00	1
10	1	0	0	1	1	01	1
10	1	0	1	0	0	00	1
10	1	0	1	1	1	01	1
10	1	1	0	0	1	11	1
10	1	1	0	1	1	xx	0
10	1	1	1	0	1	11	1
10	1	1	1	1	1	xx	0

If repeated attempts at reconfiguring fail, then the fault is determined to be a hard fault. Then the fourth unit is brought into service until the system can find a higher-level correction. Even if no further repair is possible, this approach will autonomously repair up to one failing logic block per cell. Therefore, it can tolerate many (up to the number of cells in the system) simultaneous soft/hard faults (assuming no more than one failing block per cell). To summarize, the enhanced-QMR benefits are presented as follows:

- Automatically corrects the vast majority of transient faults (like TMR)
- Uses BER measurement to distinguish transients from soft or hard defects
- Automatically identifies the soft/hard-failing unit and reports it for eventual repair
- Autonomously replaces the soft/hard-failing unit in order to maintain TMR operation
- Continues to run in TMR mode while a soft defect is repaired/healed
- Automatically tolerates up to one hard or soft defect per cell

Conversely, the enhanced-QMR limitations is discussed as follows:

- Repair/healing of multiple hard faults requires assistance from higher levels of the system (not fully automated)
- Overhead is 300% (plus the cost of error-reporting logic)

At the lowest level, redundant logic blocks within a cell are programmed for self-test and for isolating faults to a single failing logic block. The redundant blocks autonomously correct transient errors in real-time, without system interruption (online self- test/repair). The approach can tolerate a large number of independent, simultaneous such transient errors. In addition, soft errors are autonomously detected and repaired by identifying the failing logic block and reprogramming just it using partial reconfiguration techniques. This dynamic partial reconfiguration also can be accomplished online, without functional interruption of

the system. For the very infrequent case of a detected hard error (i.e. one that resists repair by reprogramming), we resort to the isolated reconfiguration of the failing cell, using a preprogrammed spare logic block judiciously located adjacent to each functional cell in the layout.

Last but not least, it is worth mentioning that the above methodologies can also be applied to ASIC designs. However, the healing feature which requires reprogramming/reconfiguring is restricted to FPGA applications. These strategies can be deployed at the block level, module level, and system level. In most cases, as we will see in the experimental results, the enhancements overhead is negligible compared with that of the redundant modules, and the performance penalty is also minimal. This low level design strategy combined with high level autonomous partial reconfiguration tremendously increases the system reliability.

3.4 State Synchronization techniques

Traditional redundancy based methods cannot restore the sequential logic state in the faulty logic, the data stored in the flip-flops and registers will be lost when switching to a spare logic block. Similarly when the repaired module is switched back online, the internal sequential states are not initialized. Therefore, for a modular redundancy technique based self-repairing architecture, a state synchronization mechanism between the repaired module and it's neighbors is required.

One way to ensure the correctness of a sequential state is to use majority voters at the input to the register, as illustrated in Figure 3.5. Data stored in the sequential state register is not directly assigned by the combinational logic block of that module. Instead, combinational logic blocks of all three redundant modules must first vote to generate the output that represents the majority of the redundant logic blocks. This voted result can then be assigned to each sequential state register. Since the register state is the majority of three logic blocks, data stored in the register is correct as long as no more than one module is

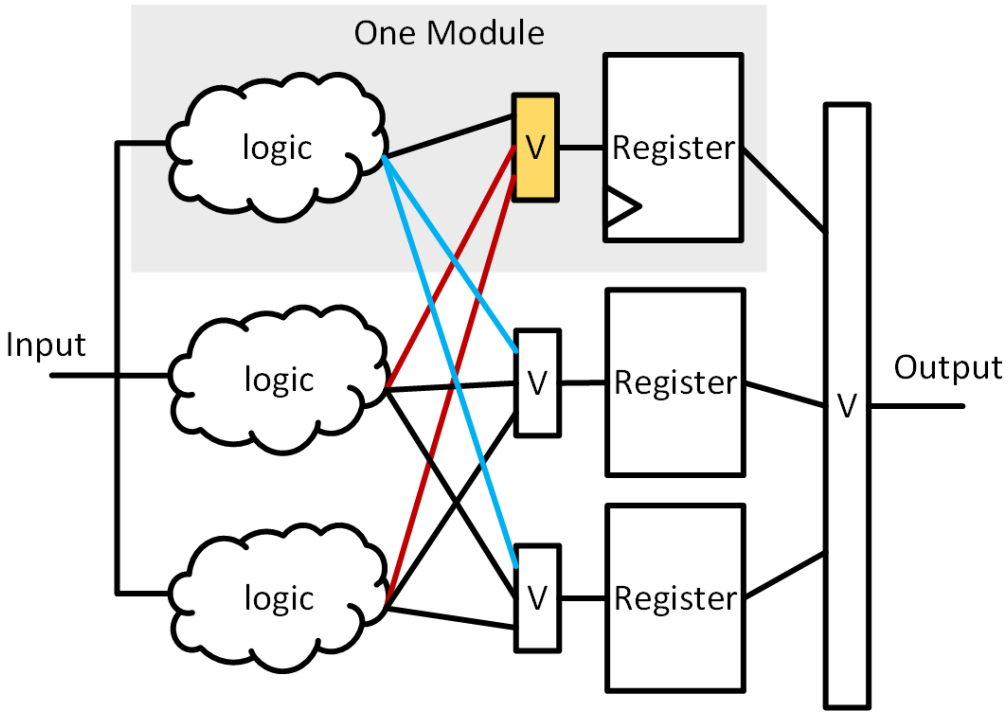

Figure 3.5: Traditional method for TMR state synchronization.

failing. However, the voter only works for an odd number of modules. An even number of modules could potential create a scenario where exactly one half of the modules are different from the other half, thus no result can represent the majority.

Alternatively, we propose a method to synchronize the registers by selectively rerouting every state register inputs. As presented in Figure 3.6, a multiplexer with one-bit control signal is added to switch the inputs of the state register. This one-bit control signal, labeled "sync" in the diagram, is generated by the fault management unit, which is discussed in the next section. One input of that multiplexer, named "default input", is the output of this module's combinational logic. This input will be selected during the normal operating mode (sync signal is zero). The other multiplexer input is the output of the nearby redundant module's combinational logic. This input is selected when the state synchronization is started (sync signal is one). This approach works for any number of redundant modules. In addition, it also costs less area overhead compared with traditional state synchronization technique since each module requires only one extra input, one extra output and one multi-

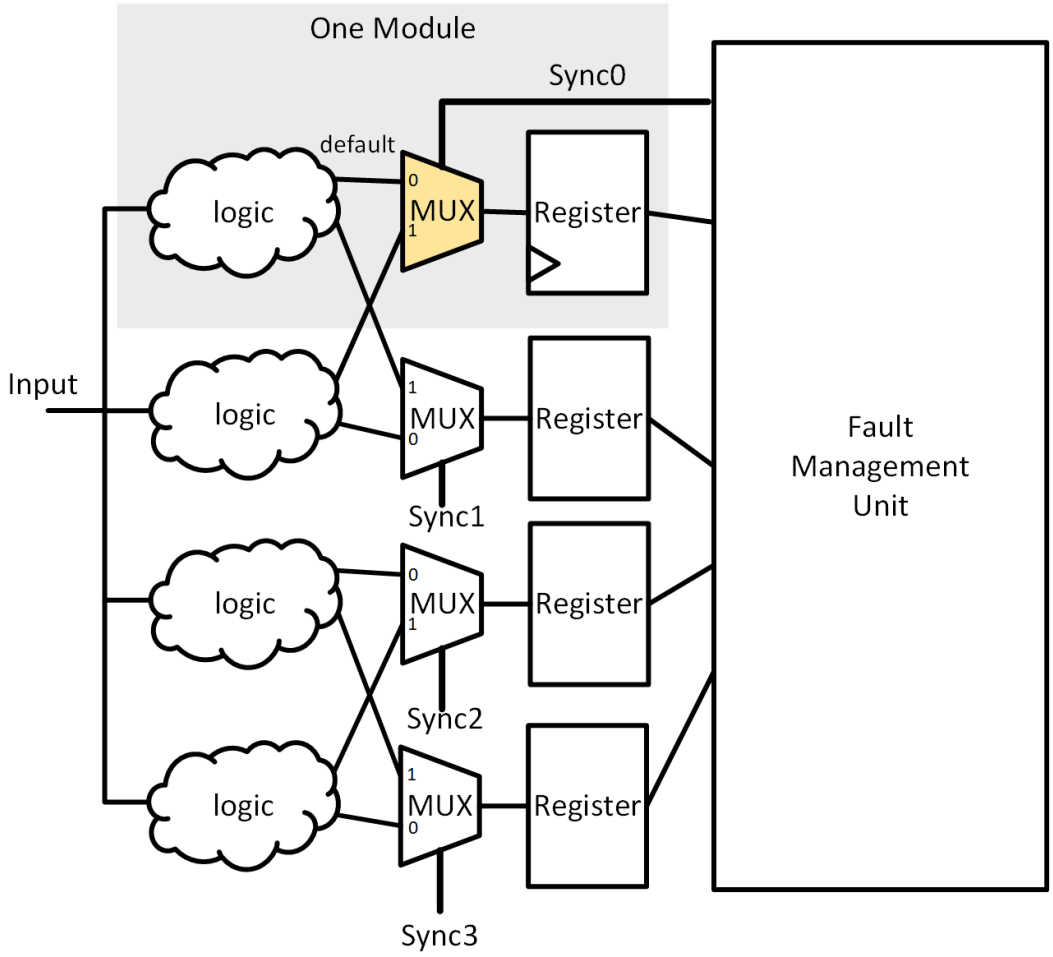

Figure 3.6: State synchronization technique for self-repairing system. The interconnection between the combinational logic and the multiplexer for the rest modules is similar to the highlighted first module and is not shown in this diagram.

plexer with one-bit control signal. All the logic states of the recently activated module will be synchronized to the nearby module in one cycle.

Figure 3.7 presents the structure of the fault management unit. As shown in the figure, the fault management unit is primarily consisted of an enhanced QMR voter with the error code register, a BER counter, a decoder and some flip-flops (4-bit register). The 2-bit latched error code is connected to a decoder and the latched outputs are the four "sync" control signals, labeled sync0 to sync3. Each of them controls the corresponding state synchronization multiplexer in the redundant module. By default, the outputs of the 4-bit register are all zeros, indicating that all multiplexer select the default input. However, if a

soft or hard error is detected, the error code will activate the corresponding output. In this example, F1 is the faulty module, thus the second output of the decoder is marked as 1 on the output. As a result, the Sync1 is high while the remainder are still zero. Consequently, the multiplexer in module F1 will select the neighbor's input at the rising edge of the unlock signal. The unlock signal is generated by the error reporting interface. It is used to indicate that the healing procedure is finished. Details about the unlock signal, error reporting interface and the healing controller are discussed in the later chapter.

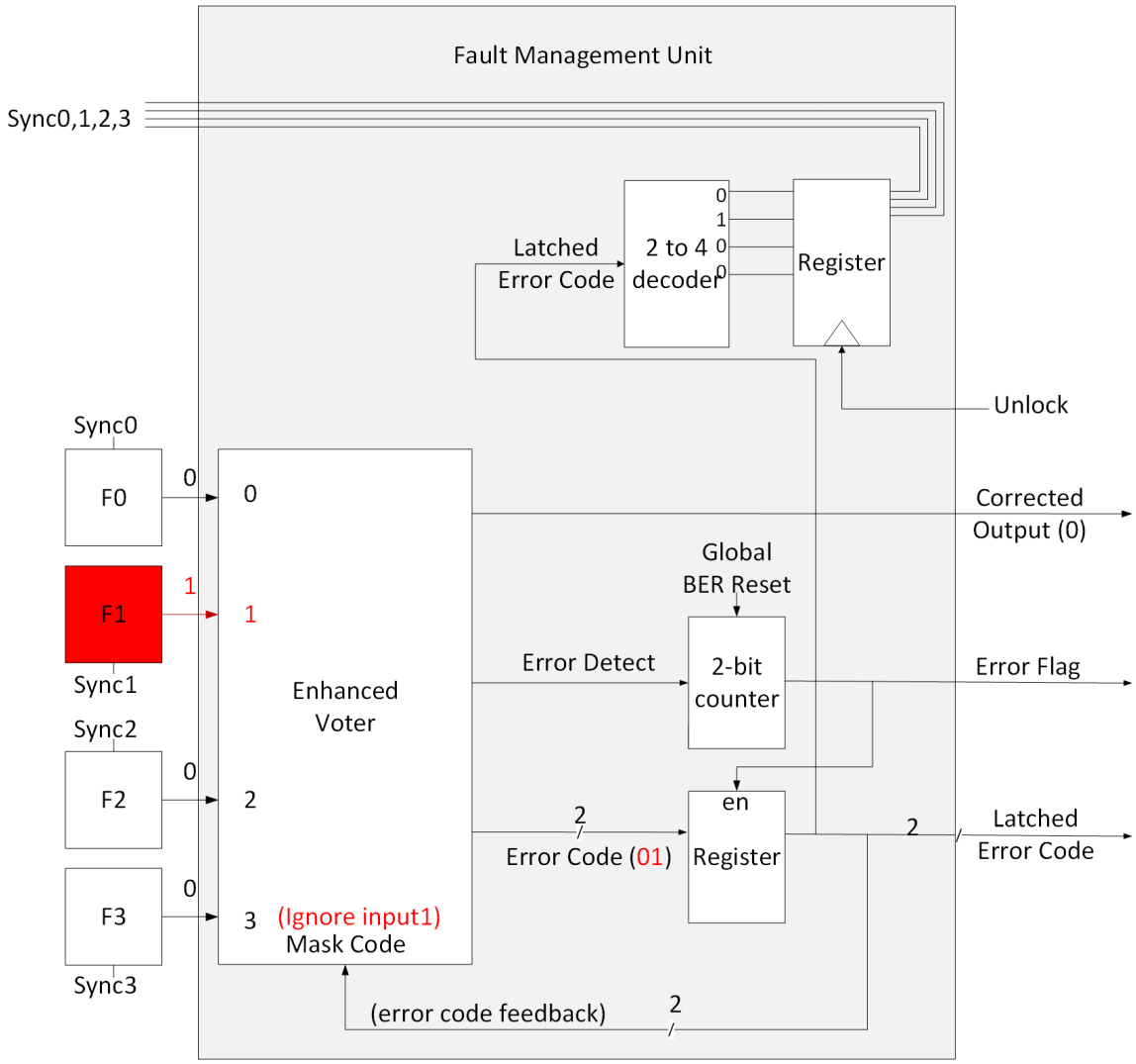

Figure 3.7: Fault Management Unit diagram.

The state synchronization technique can also be used as a healing method for failures

caused by state-bit corruption. In that case, partial reconfiguration is unnecessary since the reconfiguration is primarily focus on healing the permanent defects in combinational logic (configuring memory in FPGAs). First applying a state synchronization technique and then checking if the module is healed is more efficient than directly applying partial reconfiguration. The healing process reuses the same self-testing hardware therefore no extra resource is required.

CHAPTER 4
HEALING CONTROLLER

The healing controller that manages the repairing procedure is introduced in this chapter. The FPGA dynamic partial reconfiguration technology [47] is used to heal the faulty module. In order to understand the dynamic partial reconfiguration technique, we need to first understand how the FPGA works. In the first section, an introduction about FPGA working principles is presented. The next section discusses the details of the dynamic partial reconfiguration technique. The last section presents the design and implementation of the healing control system.

4.1 FPGA working principle

FPGA architecture

The typical architecture of an FPGA [48] is illustrated in Figure 4.1. An FPGA essentially is an array of logic gates that can be arbitrarily interconnected together to make a customized circuit. The term "gate" in Field Programmable Gate Array is not strictly accurate. The basic cells of FPGA are not the basic NAND or NOR gates but more sophisticated digital sub-circuits called configurable logic block (CLB). As presented in Figure 4.2, the CLB consists of several lookup tables (LUTs), flip-flops, and routing controls such as multiplexers. The lookup tables are programmed to implement the user combinational logic functions. The flip-flops complete the functionality of CLBs and are used for sequential state storage. The multiplexers are applied to customize the CLB internal routing.

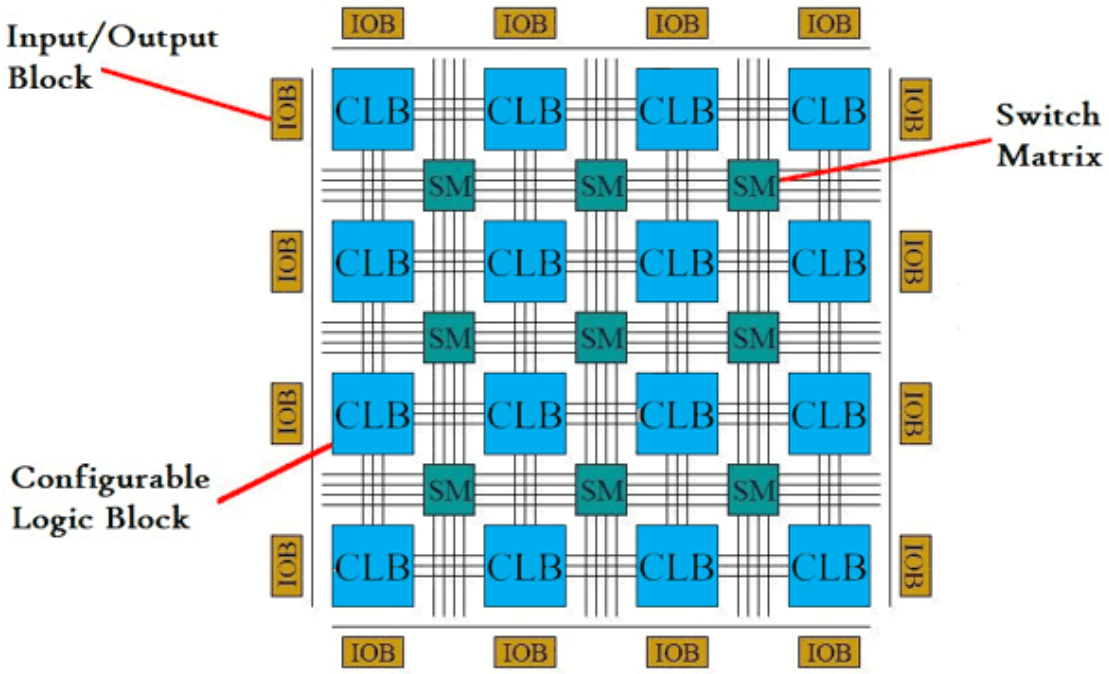

Figure 4.1: FPGA system architecture [48].

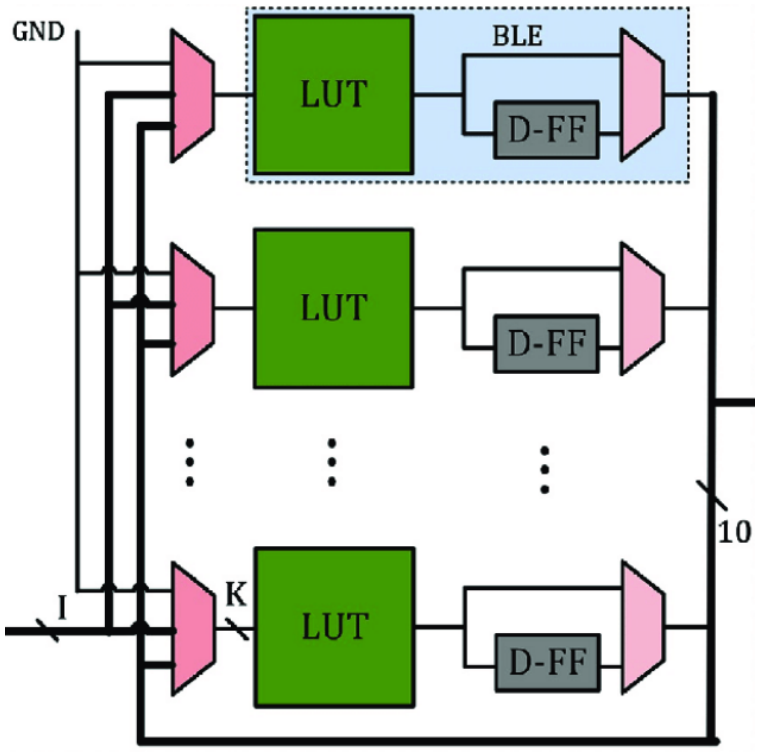

Figure 4.2: Configurable Logic Block architecture [49].

Each CLB only has a limited number of resources. Therefore, in order to implement a large digital design, many CLBs are interconnected. For this reason, a matrix of interconnect switches is used in FPGA. As presented in Figure 4.1, the switch matrix contains the transistors to turn on/off connections between different routing lines. Thus not only the CLBs but also the interconnections can be "programmed".

In order to connect the integrated circuit to external circuits, Input/Output pads or IO banks are used. These consist of pull-up/pull-down resistors, buffers, and inverters, facilitating the communication between the FPGA internal logic function and the peripheral components on the board.

Besides the basic elements discussed above, modern FPGAs also have built-in hard blocks such as Memory controllers, digital signal processing (DSP) blocks, high-speed communication transceivers, PCIe Endpoints, etc. All these hardware components establish a hardware logic layer that provides the necessary elements to form a circuit. The reprogrammability is supported by the configuration layer which stores the FPGA configuration information through a binary file called bitstream or bit file, as shown in Figure 4.3.

This binary file contains all the information that determines the implemented circuit, such as the values or the truth table stored in the lookup tables, the initial values for the flip-flops and memories, the routing information for the switch matrix, and the voltage standards for the IO pins. Therefore, the function implemented by the hardware logic layer is programmed by the values stored in the configuration memory. Most modern FPGA configuration memory is based on Static Random Access Memory (SRAM) and are hence volatile. This facilitates the implementation of FPGA design by simply downloading a bit file. Similarly, FPGA design modification only requires changing the contents of the configuration memory or rewriting a new bit file. This operation is called FPGA configuration and can be performed through external FPGA interfaces such as Joint Test Action Group (JTAG), SelectMap [50] or internal interface such as Internal Configuration Access Port (ICAP) [51].

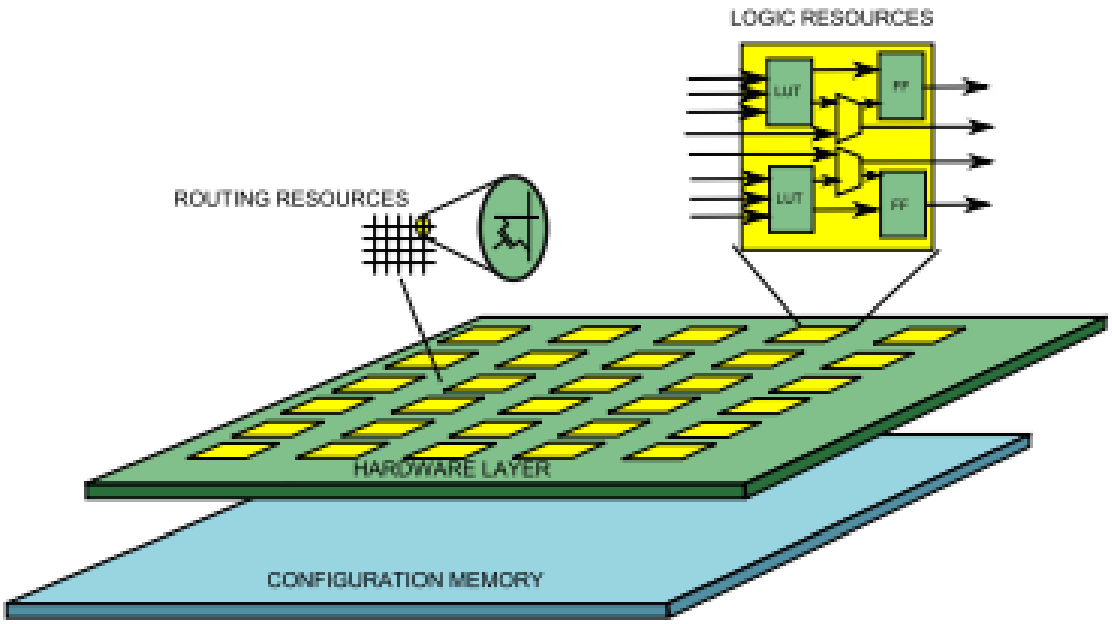

Figure 4.3: FPGA architecture with configuration memory layer and hardware logic layer [48].

FPGA programming

The process of designing and implementing an integrated circuit design on FPGA is called FPGA programming. The complete process includes building the design using HDL (Hardware Description Language) code such as Verilog or VHDL, converting the HDL code to a basic FPGA components formed circuit and generating this output file in binary format that FPGAs can understand, and programming the output file to the physical FPGA device using programming tools.

4.2 Partial Reconfiguration

As presented in the previous section, unlike a fixed application specific integrated circuit (ASIC), FPGA technology provides the flexibility of reprogramming without going through re-fabrication process. Partial Reconfiguration (PR) takes this flexibility one step further. As presented in Figure 4.4, this architecture has multiple configuration layer. Since each

configuration layer is independent, modification of one or more configuration layers would not affect the content stored in other layers. Therefore, it allows a predefined portion of an FPGA, also known as the reconfigurable region, to be reprogrammed while the remainder of the device continues to operate. The file to configure the FPGA is usually in the format of a binary file. [47]. We use "configuration file" or "BIT file" interchangeably in the following discussion. Figure.4.5 provides a high-level overview of the mechanism behind

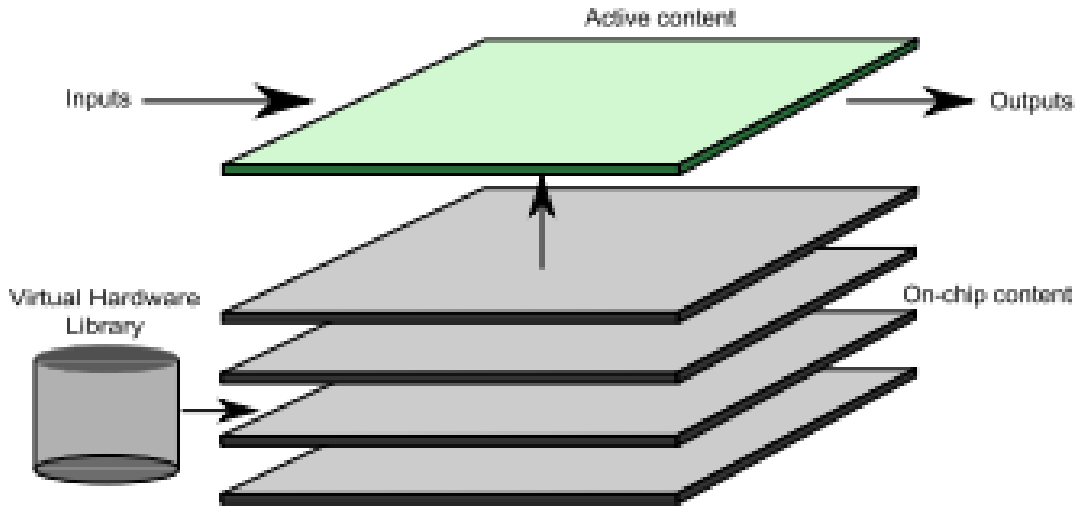

Figure 4.4: FPGA architecture with multi-configuration layer [48].

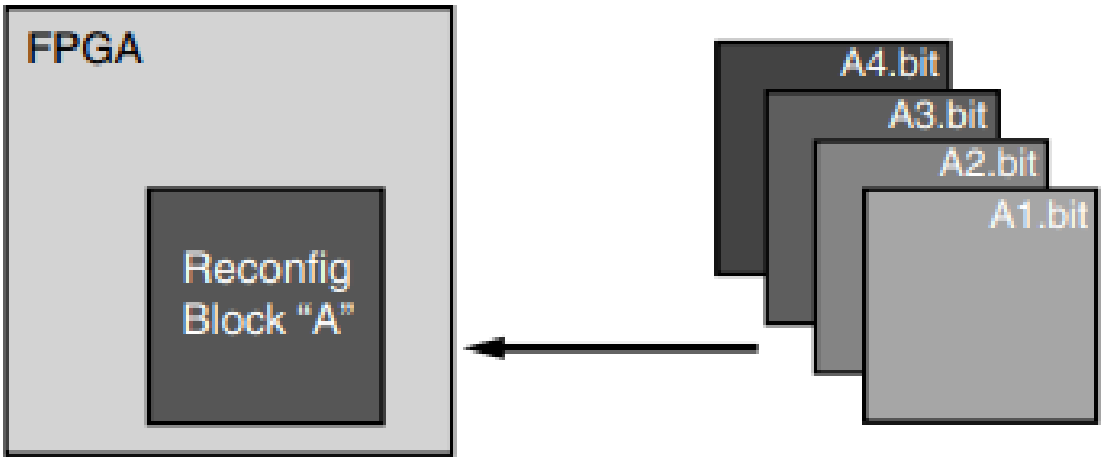

Figure 4.5: How Partial Reconfiguration works [47].

Partial Reconfiguration. In the original design, the FPGA blocks are divided into two different classes. The gray area of the FPGA block represents static logic and the black area represents reconfigurable logic. The granularity of the reconfiguration region is related to the configuration memory frame. Details about the reconfiguration region and the configuration memory frame is discussed in the later section of this chapter. In this example, block A is marked as a reconfigurable region. The function implemented in this region can be overwritten by downloading one of several partial BIT files, e.g. A1.bit, A2.bit, A3.bit, or A4.bit while The static logic remains functioning and is unaffected by the loading of a partial BIT file.

PR Architecture

Xilinx has been supported PR for many years. Early age products, such as the XC6200 series [52] FPGA, contain only a single configurable memory layer. Later products, such as Virte-II, Virtex-4, Zynq and Ultrascale, achieved partial reconfiguration based on multi configuration memory layers [53][54][47].

For Virtex-II [53] series, Xilinx organizes the FPGA resources in a columnar fashion. These primitives include configurable logic blocks (CLBs), Block RAMs, and multipliers. The configuration memory is thus also organised in column, in fact each frame is 1-bit wide and whole FPGA device height and configures a narrow vertical slice of many physical resources [55]. Xilinx groups them into several different configuration columns, such as Input/Output Block (IOB), CLB and BlockRAM, depending upon their role. Combine several frames from one or more classes construct a partially reconfigurable region. Since the height of each frame is the full height of the FPGA, each reconfiguration region is also restricted to the full height of the FPGA, as shown in Figure 4.6. Therefore it is not efficient in hardware utilization but relatively easy for floor-planning. This architecture simplifies the routing process, thus the run time circuit relocation is not an issue.

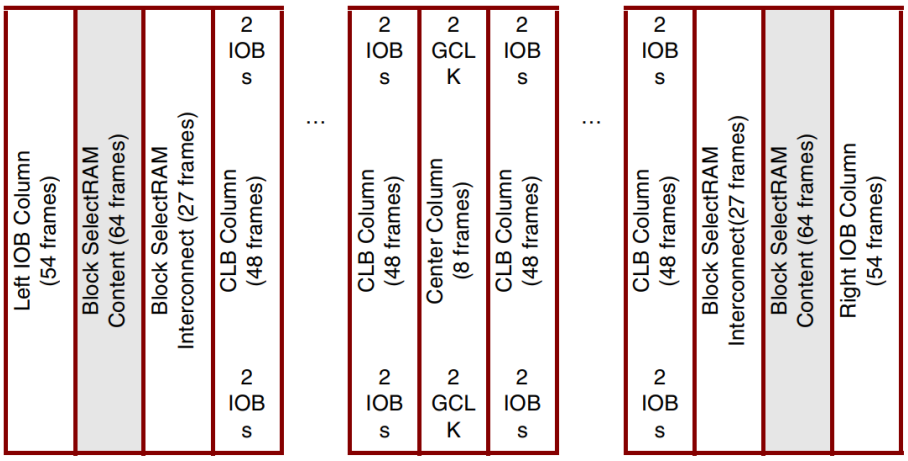

Figure 4.6: Partially Reconfigurable regions in Xilinx Virtex XCV50 FPGA.

In order to support context switching between different partial configuration memory files, every circuit targeted for the same partially reconfigurable region must have the same interface to the static (non-PR) region. In Virtex-II devices, this is achieved by fixing the routing between the static and PR regions and managing the connectivity with internal tri-state buffers (TBUFs). To support runtime circuit relocation, the relative positions of these TBUFs also need to match among different PRRs. The limited number and fixed position of TBUFs available on the chip further restricts the size and positions of PRRs.

For Virtex-4 family of FPGAs [54], TBUFs were replaced by bus macros [56], as shown in Figure 4.7. These bus macros are constructed by lookup tables. Since the number of lookup tables are large and distributed throughout the FPGA, as opposed to the limited number and fixed locations of TBUFs, this increases the flexibility of connectivity arrangement. Moreover, The size of frames was also reduced in the Virtex-4. Unlike the Virtex-II, where frame size is dependent on device size, it is 1 bit wide and 16 CLBs high and contains 41 32-bit words (1312 bits). The reconfigurable region thus also no longer has the restriction of being full height of the device, but rather must be a height that is a multiple of 16 CLBs. Because of this modified architecture, the floorplanning task becomes a two dimensional problem instead of one dimensional problem, which also increase the difficulty for runtime relocation.

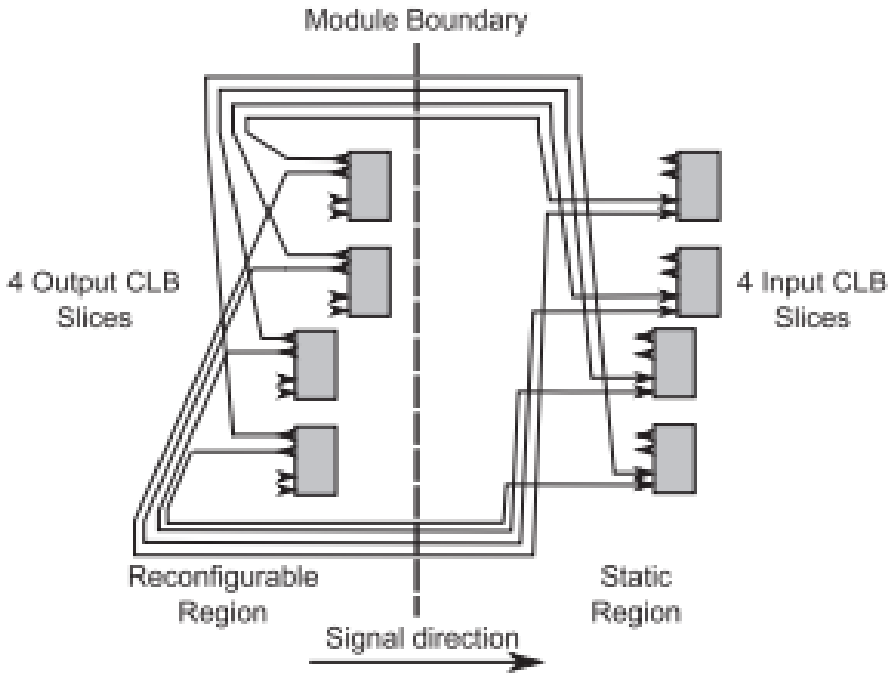

Figure 4.7: A bus macro showing the connectivity between the static region and a reconfigurable region [48].

Start with the Virtex-5 series, Xilinx divided the device into several rows and columns as shown in Figure 4.8.

Each row represents a clock region and each column, also called block, contains a single class of FPGA resource as presented in the previous section. The intersection between the row and the collum is called tile. Depending on the primitives of the block, the tile is categorized into CLB tiles, DSP tiles, BRAM tiles etc. They are the basic unit for configuration frame and partially reconfigurable region. One CLB tile contains 20 CLBs, one DSP tile contains 8 DSP slices, and one BRAM tile contains 4 Block RAMs. Virtex-6 and Xilinx 7-series FPGAs (Artix, Kintex, and Virtex-7) also have a similar tile architecture, the main difference is the number of resources each tile contains.

These advanced architecture improves the efficiency of hardware utilization. It also increases the flexibility of FPGA implementation and enables multiple PRRs with varying sizes with different kinds of resources. However, all these architectural improvements also

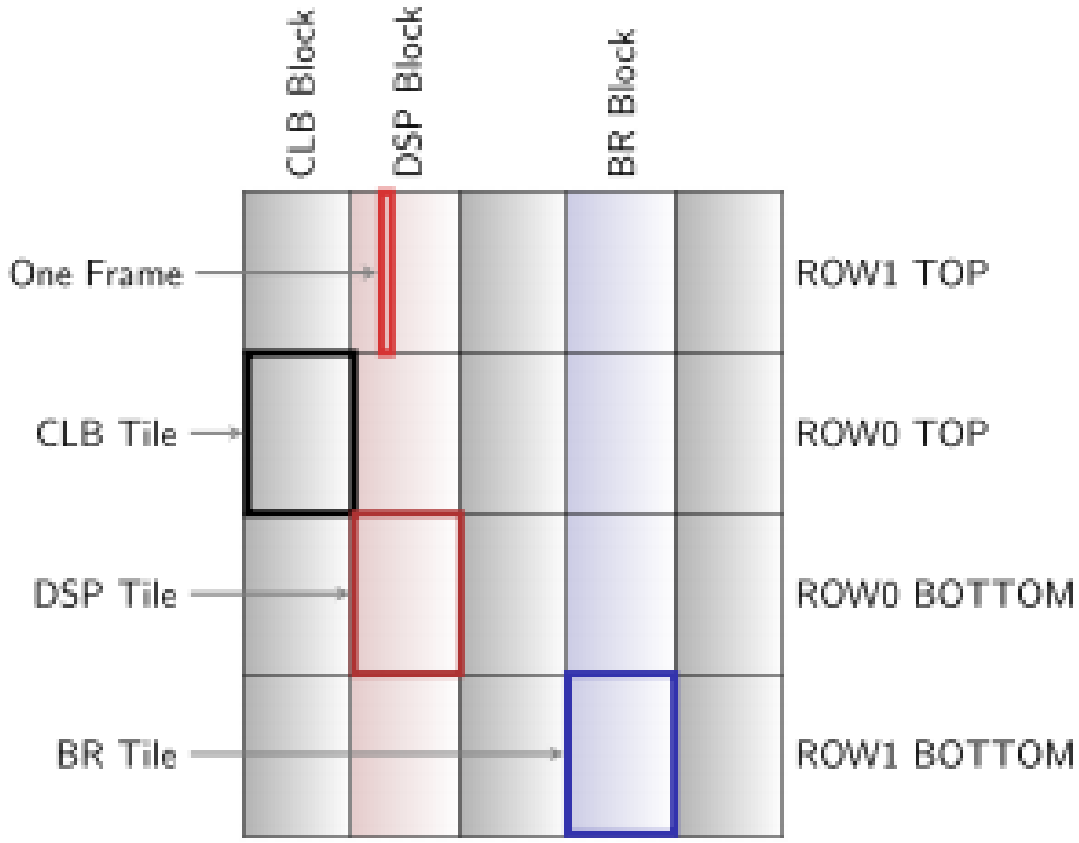

Figure 4.8: Xilinx 7-Series architecture [48].

dramatically increase the difficulty of runtime circuit relocation.

PR Flow

The design and execution of partial reconfiguration can be achieved by Xilinx Vivado Design Suite [47]. The final hardware design is composed of two parts, the static region and one or more reconfigurable regions (PRRs).

The static region is the portion of the design, which does not change its functionality during system operation. PRRs implement the reconfigurable modules, and can be reconfigured at runtime. A single reconfigurable region can implement many modules in a time multiplexed fashion; all reconfigurable modules implemented in the same PRR constitute a reconfigurable partition. The design flow is illustrated in Figure 4.9, the first step is to syn-

thesise the static and reconfigurable modules separately using Xilinx tools or third-party synthesis tools. The second step is to define the reconfigurable regions and allocate the corresponding module to them (partitioning). Notice that it is the designer's responsibility

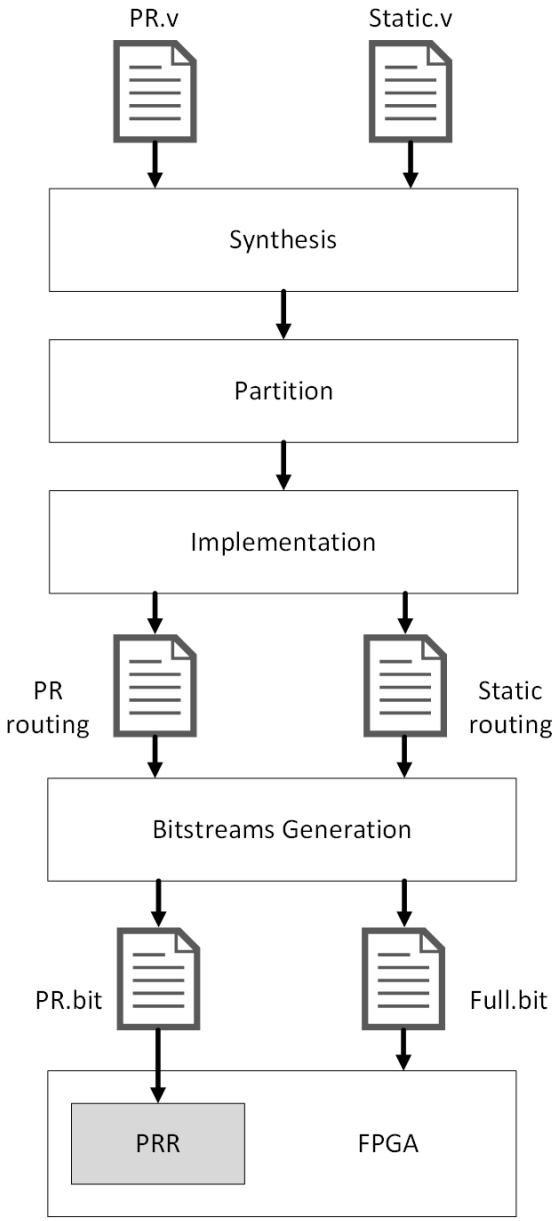

Figure 4.9: Partial Reconfiguration design flow.

to verify the floorplanning and ensure that there is no violation against the reconfigurable regions requirement in the FPGA fabric. For example, the height of the regions must be that of a multiple of 16 CLBs and should be aligned to clock region boundaries as explained

in the partial reconfiguration architecture section.

The next step is to implement the static design. This is achieved by first using a reconfigurable partition as a placeholder and implementing the complete system. Then the reconfigurable regions are removed, leading to a placement and routing file only for the static region. After the static design implementation is finished, it preservers that area for all other configurations.

The following step is to implement the reconfigurable design. Since the static region is locked, adding the reconfigurable partition to the static design and implementing this whole system would only generates the valid placement and routing file for the partially reconfigurable region. If multiple reconfigurable partition exist, each reconfigurable partition needs to be added to the static design and implemented. This step would be repeated until all the reconfigurable partitions are implemented. Designers do not need to worry about the interface between the static region and partially reconfigurable regions, since Vivado would automatically implement the interface logic.

Finally, the tool generates a full configuration file as well as partial bitstreams for each PRR and each configuration. The FPGA is initially configured using one of the full bitstreams and the PRR can be reprogrammed using a any corresponding partial bitstream at run time without interrupting the remainder systems operation.

4.3 Healing controller program

The healing controller is a program executed on a personal computer. This program utilizes the Vivado Design Suite to manage the partial reconfiguration bitstream downloading procedure. When the healing controller receives the repairing request and the faulty module identity from the error reporting module of the FPGA, it generates the partial reconfiguration command in order to repair the faulty area.

The error message format is presented in Figure 4.10. The total length for each error message is two bytes. The first bit represents the error flag, followed by a two-bit faulty

module identity. The next five bits represent the output identity of a cell. The last eight bits indicate the cell ID. When the error flag is active, an intermittent error is detected. Therefore, the system requests for a partial reconfiguration.

1 bit	2 bits	5 bits	8 bits
Err Flag	Module ID	Output ID	Cell ID

Figure 4.10: Error message format.

For example, assume an FPGA design contains two cells, cell A and cell B. Cell A has two outputs while cell B has three outputs as illustrated in Figure 4.11. The faulty module ID, output ID, cell ID, the corresponding redundant module name and the partially reconfigurable region is listed in Table. 4.1.

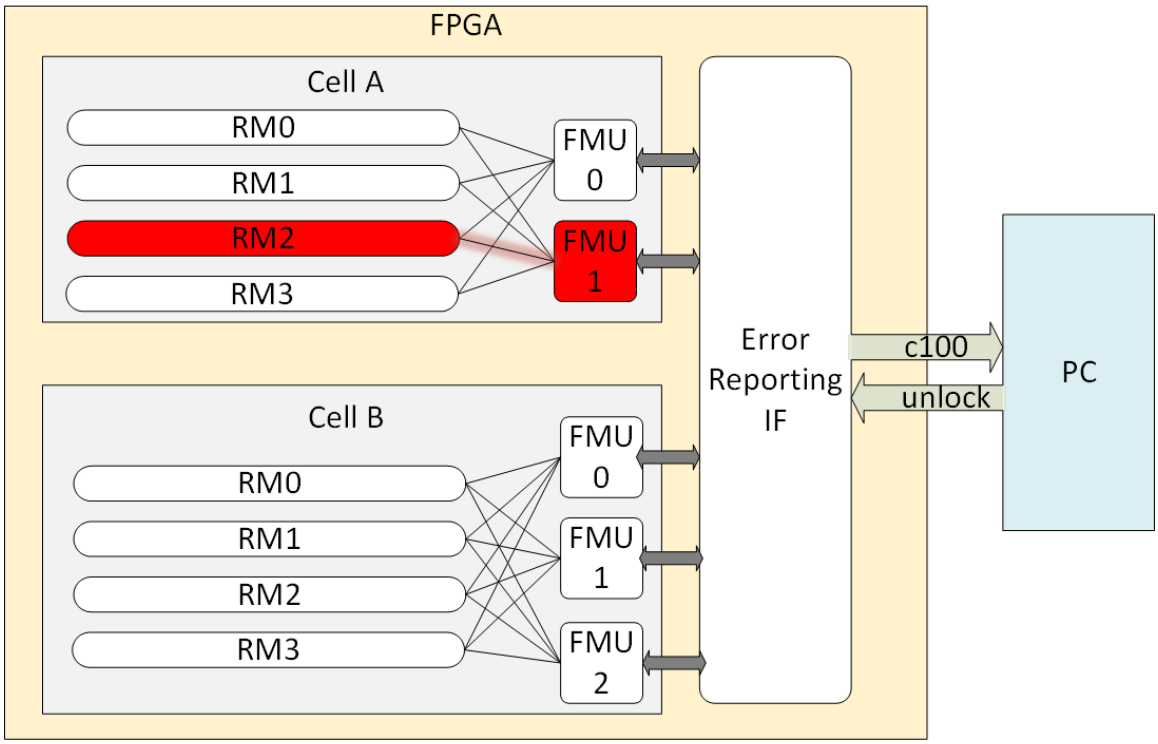

Figure 4.11: Example of the interaction between the healing controller (in a PC) and the FPGA when a soft error is detected.

Table 4.1: Relationship between the faulty module ID, redundant module name and the partially reconfigurable region.

Module Name	Output ID	Module ID	Cell ID	PRR
RMA_0	x	0	0	PRR_A0
RMA_1	x	1	0	PRR_A1
RMA_2	x	2	0	PRR_A2
RMA_3	x	3	0	PRR_A3
RMB_0	x	0	1	PRR_B0
RMB_1	x	1	1	PRR_B1
RMB_2	x	2	1	PRR_B2
RMB_3	x	3	1	PRR_B3

Figure 4.12 presents the pre-defined partial reconfigurable regions. If the RMA2 module collapses and output 1 reaches the error threshold, the error message is generated as **1100000100000000** in binary (or **C100** in hexadecimal). The first **1** indicates an error flag, the following **10** and the last eight bits **00000000** identify the faulty module, which is the "redundant module 2" of cell A. The output ID **00001** presents the error message is created by output 1. Even though this output ID does not make a difference in this example, when multiple fault management units of different outputs catch errors, this output ID is required to choose which module should be repaired first. In other words, when more multiple modules of the same cell fail simultaneously, the output ID determines the priority of reparation. According to Table. 4.1, the partial bitstream PRR_A2 will be downloaded to program the partial reconfiguable region PRR_A2 in Figure 4.12.

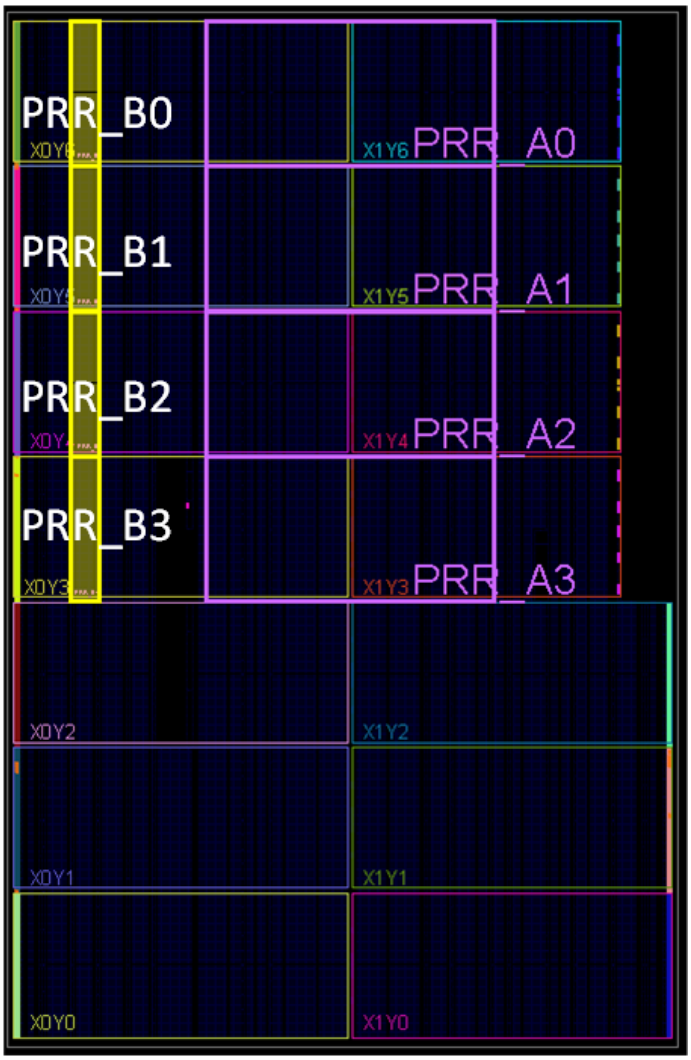

Figure 4.12: Example of the pre-defined partial reconfigurable regions.

Figure 4.13 presents the flowchart of the healing controller program. First, the program reads the faulty module identification from the serial port (a serial communication interface through which information transfers in or out one bit at a time in contrast to a parallel port). Second, depending on the value or the id, the healing controller downloads the corresponding partial reconfiguration bit file, which are generated from the partial reconfiguration project development process discussed above, to the FPGA. Finally, it transmits a “unlock” signal through the serial port and waits for the next serial message.

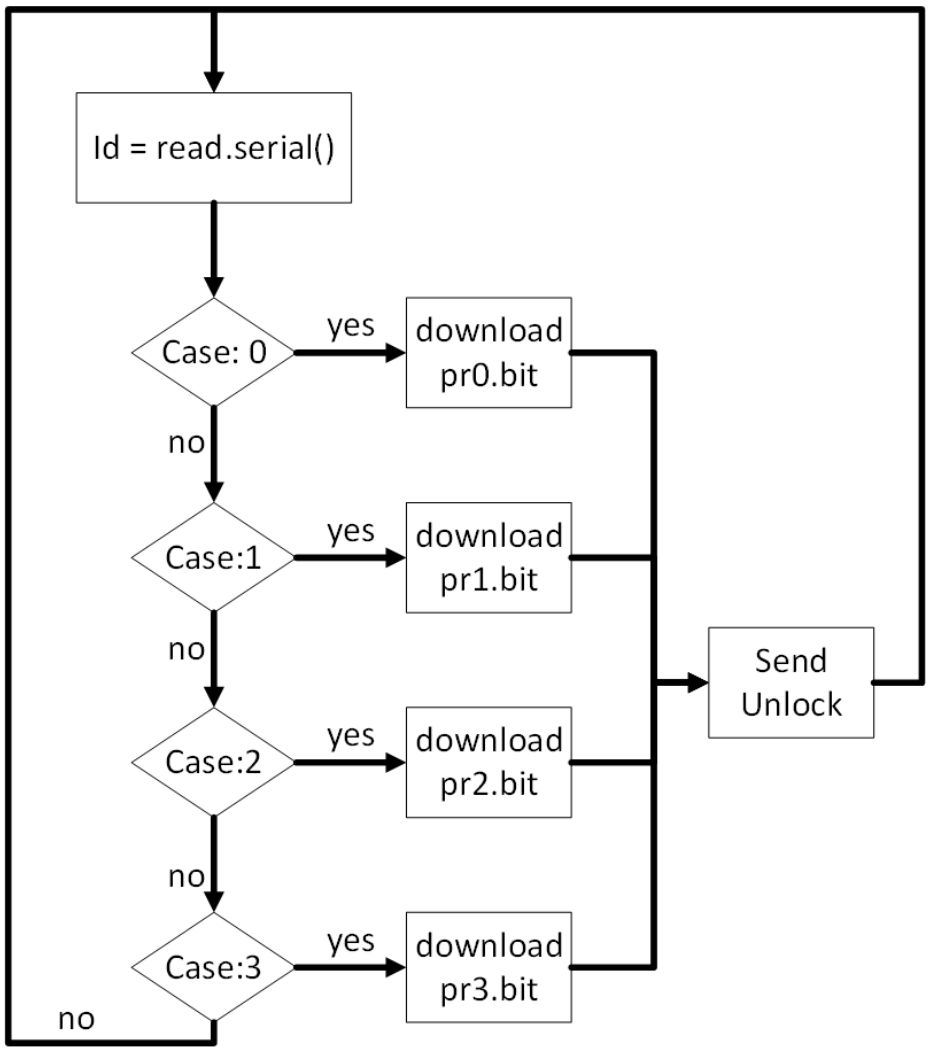

Figure 4.13: Healing Controller flowchart.

CHAPTER 5

SOFTWARE TOOLCHAIN

In the previous chapters we described how the self-repairing system architecture is capable of detecting, identifying and correcting errors autonomously, yet the transformation from the original source file to the self-repairing version was not discussed. This chapter provides a method to automatically complete this transformation.

5.1 Introduction

The goal of the software stack is to provide an autonomous tool for deploying the self-repairing architecture on any digital design. Since Verilog HDL is a widely used design language to express the fabric of a hardware structure for both ASIC and FPGA-based circuits, the challenge of autonomously deploying the self-repairing architecture is equivalent to automatically change the original Verilog code to the self-repairing version. For this reason, we develop a HDL converter, as illustrated in Figure 5.1.

Figure 5.1: HDL converter.

For example, considering the Verilog implementation of a full-adder presented in Figure 5.2, by replicating the user logic block and inserting the enhanced fault management unit described in Chapters 3, the equivalent circuit implemented in Verilog language is presented in Figure 5.3.

```
module fulladd(s, c_o, a_i, b_i, c_i);
   output s, c_o;
   input a_i, b_i, c_i;
   assign s = (a_i^b_i)^c_i; // sum bit
   assign c_o = (a_i & b_i) | (b_i & c_i) | (c_i & a_i); //carry
      bit
endmodule
```

Figure 5.2: Original HDL design file example: full adder

```
module fulladd_conv (
//input output port declarations
   output s,
   output c_o,
   input a_i,
   input b_i,
   input c_i
);

// Internal connection
   wire [3:0] s_qmr, c_o_qmr;
   wire [1:0] err_code0, err_code1;
   wire err_flag0, err_flag1;

// Instantiate redundant module
   fulladd rm0 (s_qmr[0], c_o_qmr[0], a_i, b_i, c_i);
   fulladd rm1 (s_qmr[1], c_o_qmr[1], a_i, b_i, c_i);
   fulladd rm2 (s_qmr[2], c_o_qmr[2], a_i, b_i, c_i);
   fulladd rm2 (s_qmr[3], c_o_qmr[3], a_i, b_i, c_i);

// Fault Management Unit
   qmr_enh inst0 (s_qmr, s, err_code0, err_flag0);
   qmr_enh inst1 (c_o_qmr, c_o, err_code1, err_flag1);

endmodule
```

Figure 5.3: Converted HDL file with self-repairing architecture.

Note that this example does not include the error reporting module. If the healing process is desired, it can also be included into the converted HDL file by instantiating the error reporting module. Details implementation of the the enhanced modules, such as the qmr_enh module in this example, are presented in Appendix.

Recall from the self-repairing architecture, the fault management unit and the error-reporting module must adapt to the user logic. Fortunately, most of the structure remains unchanged while only the ports related logic requires a modification. Therefore, the first important task is to identify the ports of the original design. Since the port declaration is usually on the top of the module and begins with either the keyword "input", "output" or "inout", this particular pattern can be easily searched using the Regular Expression [57]. A regular expression, also known as a "regex", is a sequence of characters that specifies a pattern. Table. 5.1 presents some example of using regular expression to match certain strings patterns. It is extremely useful in extracting and replacing information from text that takes a defined format, such as dates, urls, phone number, email addresses and even code. Therefore, it is possible to perform advanced text manipulation.

Table 5.1: Example of using regular expression to match strings patterns

Regex	**Matches any string that**
hello	contains {hello}
^The	starts with {The}
dog$	ends with {dog}
^reg$	is exactly {reg}
a(b|c)	has {a} followed by {b} or {c}
go+gle	contains{gogle,google,gooogle,goooogle,...}

Since the previous full-adder example is a combinational circuit, no state synchronization technique is implemented. To further evaluate the regular expression, consider the example of a simple sequential circuit, a counter, presented in Figure 5.4.

```
module counter    (
    output reg [7:0] count,  // Output of the counter
    input enable,  // enable for counter
    input clk,  // clock Input
    input reset      // reset Input
);

always @(posedge clk) begin
    if (reset) count <= 8'b0;
    else if (enable) count <= count + 1;
end
endmodule
```

Figure 5.4: Example Verilog code of a 8-bit counter

Note that when applying the state synchronization method, only the second statement of the "out" register, e.g. $out <= out + 1;$ requires a modification. The first line $out <= 8'b0;$ should be unaffected since the state synchronization should not overwrite the reset behavior. In other word, the state synchronization should be applied after the repairing process, not during the system reset event.

Recall the state synchronization diagram in Figure 3.6 from Chapter 3, the equivalent design in Verilog for this up counter is presented in Figure 5.5. Even though there is only one reset statement in this example, in a real world application it can be any integer number of statements. The lack of a separate memory in the finite automata limits the pattern depth it can detect. As a result, the pattern for the reset statement would be too complex to be described in the regular expression [58]. Therefore, using the regular expression to locate the target state register is impractical for sequential logic design.

```
module counter
(
  output reg [7:0] count,
  input enable,
  input clk,
  input reset,
  input [0:0] sync,
  input [7:0] neighbor_in_count,
  output [7:0] neighbor_out_count
);


  always @(posedge clk) begin
    if(reset) count <= 8'b0;
    else if(sync) begin
      count <= neighbor_in_count;
    end else begin
      if(enable) count <= count + 1;
    end
  end

  assign neighbor_out_count = count + 1;

endmodule
```

Figure 5.5: Self-repairing version of the 8-bit counter with the state synchronization technique

Moreover, identifying the target register is only the first step of applying the state synchronization method. The second step is to modify the original code by inserting the related IO ports, registers and the corresponding statements without violating the grammar and the original functionality. This could not be achieved without the knowledge of the code structure. Therefore, we need a more powerful tool that not only understands the meaning, or semantics, of the code but also its form, or syntax. This tool is called a compiler.

5.2 Typical Compiler

A compiler is a computer program that translates a program written in one language, for example C or Java, into anther language such as the machine language (usually in a bi-

nary format). First, it needs to understand the syntax and meaning of the source language. Second, it must comprehend the rules that govern the form and content in the target language. Finally, it requires a scheme to map content from the source language to the target language [59].

In order to understand the structure and meaning of the source program, a compiler first pulls it apart and analysis each basic component. Consequently, in order to generate the target program, it puts the pieces together in a different way. As presented in Figure 5.6, the front end of the compiler performs analysis, focusing on understanding the source-language program; the back end does synthesis, aiming to map the program to the target machine.

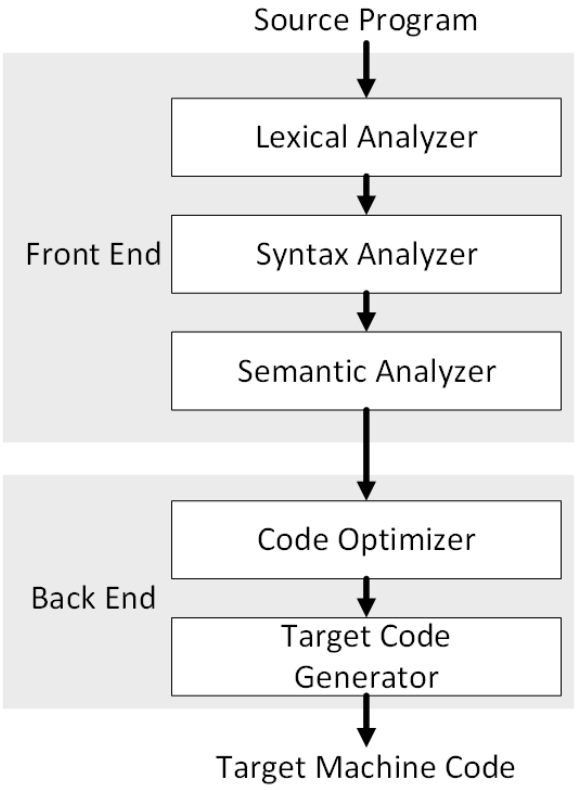

Figure 5.6: Structure of a classical compiler [60].

In order to bridge the front end and the back end, a compiler generates a special data structure for representing the program in an intermediate form whose meaning is largely

independent of the source language or the target language. Furthermore, this intermediate representation includes the knowledge of the source program obtained by the front end. To improve the translation, an optimizer is often included to analyze and rewrite the intermediate form [60].

5.3 HDL converter

Since the back end of the compiler is usually responsible for mapping a human readable code into the target machine code, it is not required by the HDL converter. Instead, we need to automatically modify the source code without violating the source language grammar. The intermediate representation is a formal data structure designed to be independent of the source languages. Therefore, it is much more convenient to be modified compared with the source program which is governed by the source language grammar. In fact, this intermediate representation allows us to inject the required IO ports, registers (for state synchronization) and the corresponding activation conditions without concerning about the grammar. Moreover, the modified intermediate representation can be transformed reversely to the source program written in Verilog. Therefore, the back end of a compiler should be replaced with an intermediate code modifier and source code generator. Details of how to generate a new program based on the existing intermediate representation is described in the intermediate representation modifier section in this chapter (section 5.3.3).

For the front end, the semantic analyzer, which focus on connecting variable definitions to their uses and checking that each expression has a correct type, can also be skipped. This is because the semantics of the language only determines what its programs mean. In other words, semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

For example, consider the pseudo code presented in Figure 5.7. This code is lexically and structurally correct. Therefore, it should not issue an error in the lexical and syntax analysis phase. However, it should generate a semantic error as the type of the assignment

differs. The “value” on the right hand side of the equal sign is a string, but it is assigned to an integer variable. These rules are set by the grammar of the language and evaluated in semantic analysis.

```
int a = ”value”;
```

Figure 5.7: Example of a syntax-correct but semantic-wrong code

Since we assume that the original HDL file is already a valid design, thus checking the meaning of the program is unnecessary. Furthermore, it is not the HDL converter’s responsibility to verify the original design. Even if the original HDL file is not valid, the electronic design automation software or the Verilog compiler will perform the semantic analysis when the Verilog code is compiled to a netlist or bitstream. Therefore, the semantic analyzer of the front end compiler is not required for the HDL converter.

As a result, compared with a typical compiler, the structure of the HDL converter is shown in Figure 5.8. First, the lexical analyzer and the syntax analyzer read the original code and generate the intermediate representation, which is an Abstract Syntax Tree (AST) in our case. Second, the AST modifier upgrades the intermediate code. Finally, the HDL code generator create a new Verilog program based on the upgraded AST.

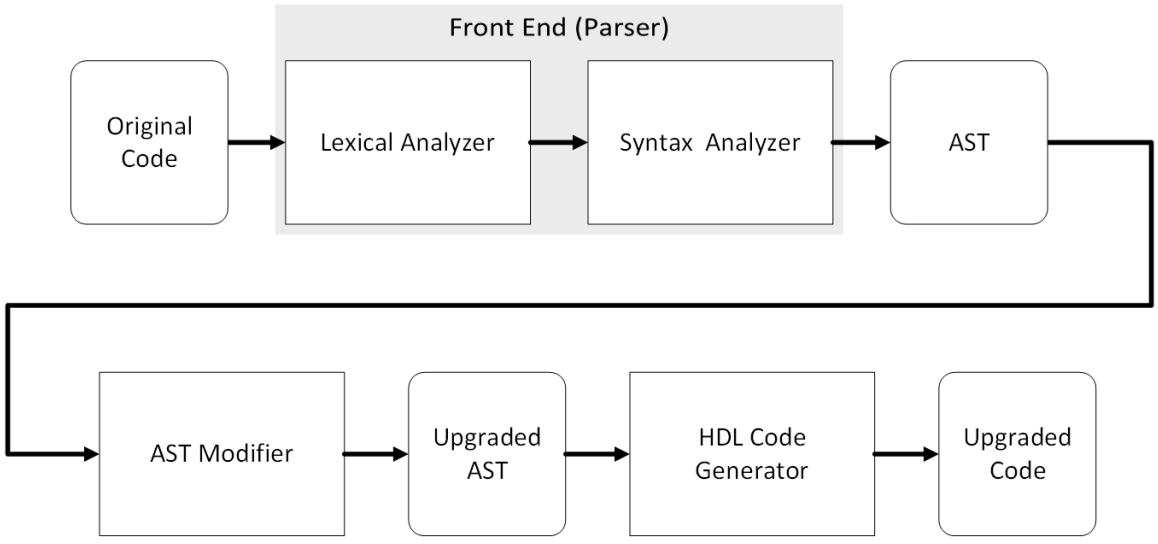

Figure 5.8: Structure of a HDL converter.

5.3.1 Lexical Analyzer

Similar to a compiler, before the HDL converter can transform one HDL file to another design, it must understand the original source program. Assuming that a computer program is a set of instructions or strings defined by a finite set of rules, called a grammar; a string is a finite sequence of symbols; and the symbols themselves are consisted of a set of finite characters. To analysis the original source codes, we start with the smallest constituent unit of the program.

Therefore, the first phase of understanding a program is to group individual characters into distinct words or symbols and classify each word with a part of speech or type. This step is called lexical analysis or scanning. The lexical analyzer reads the stream of characters and groups the characters into meaningful sequences called lexemes. As illustrated in Figure 5.9, for each lexeme, the lexical analyzer produces a token as an object with two keys: token-name and attribute-value. The token-name is an abstract symbol that is used

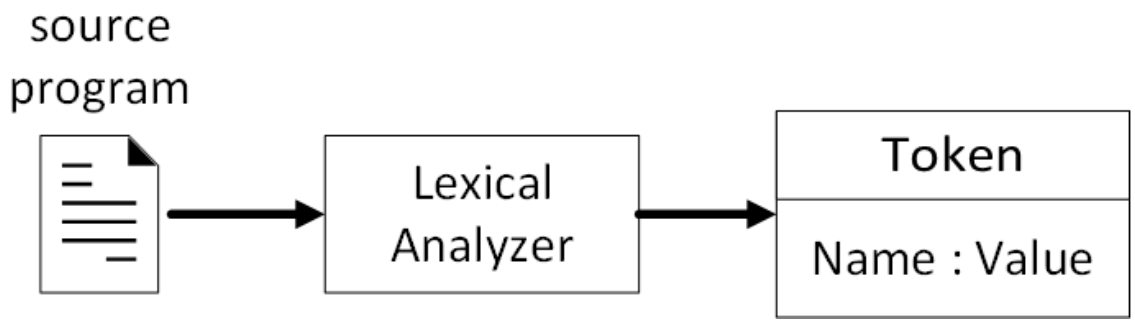

Figure 5.9: Lexical Analyzer.

during syntax analysis, and the attribute-value points to an entry in the symbol table for this token. In other words, the lexical analyzer creates a connection between the individual symbols of the source program and the predefined symbol types of the source language. As a result, each symbol is categorized into an associated type. The tokens are then passed on to the subsequent phase as a basic knowledge of the symbols. Figure 5.10 presents the output of the lexical analyzer as applied to the counter example.

```
module MODULE 1  0
counter ID 1  7
( LPAREN 1  18
output OUTPUT 2  24
reg REG 2  31
[ LBRACKET 2  35
7 INTNUMBER_DEC 2  36
: COLON 2  37
0 INTNUMBER_DEC 2  38
] RBRACKET 2  39
out ID 2  41
, COMMA 2  44
input INPUT 3  76
enable ID 3  82
, COMMA 3  88
input INPUT 4  117
clk ID 4  123
, COMMA 4  126
input INPUT 5  148
reset ID 5  154
) RPAREN 6  180
; SEMICOLON 6  181
always ALWAYS 7  183
@ AT 7  190
( LPAREN 7  191
posedge POSEDGE 7  192
clk ID 7  200
) RPAREN 7  203
if IF 8  209
( LPAREN 8  212
reset ID 8  213
) RPAREN 8  218
out ID 8  220
<= LE 8  224
8'b0 INTNUMBER_BIN 8  227
; SEMICOLON 8  231
else ELSE 9  237
if IF 9  242
( LPAREN 9  245
enable ID 9  246
) RPAREN 9  252
out ID 9  254
<= LE 9  258
out ID 9  261
+ PLUS 9  265
1 INTNUMBER_DEC 9  267
; SEMICOLON 9  268
endmodule ENDMODULE 10  270
```

Figure 5.10: Lexical analyzer result of scanning the counter.v

In order to identify the lexical token, multiple regular expressions are used. As previously discussed, the regular expression is a convenient way to match a string pattern. For example, if a pattern is described as a predefined type, then whatever string matches that pattern is a member of that type.

However, a regular expression is simply an abstract formulation. In order to implement it as a computer program, finite automata are used. A finite automaton is a finite state machine that accepts or rejects strings of a language which it defines. It has a finite set of states; edges lead from one state to another, and each edge is labeled with a symbol. One state is the start state, and certain of the states are distinguished as final states. For example, a finite automata that can detect a floating number is illustrated in Figure 5.11. The initial state is labeled as state 0. If the first character is any thing from number zero to number nine (a digit), it goes to state 1. If the next charter is still a digit, it stays at state 1. Otherwise, if it is a dot, it leads to state 2. If the following character is a digit, it reaches state 3 and stays at this stat as long as the following character is still a digit. The state 3 represents that a floating number is matched.

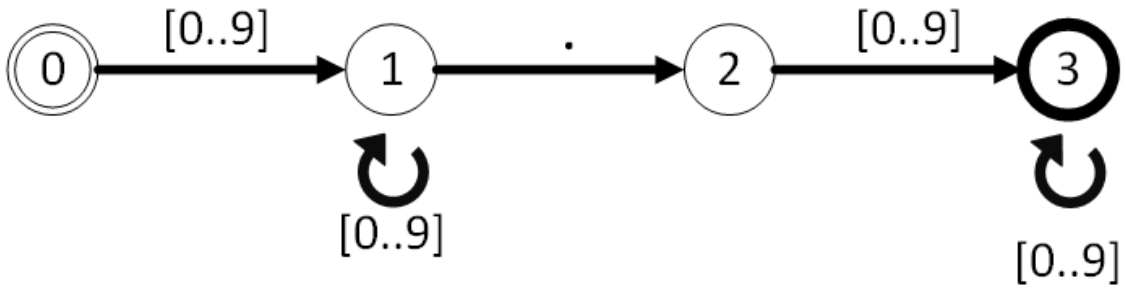

Figure 5.11: The Finite Automaton of a floating number.

5.3.2 Syntax Analyzer

After the stream of characters are grouped into words, a compiler fits the words into a grammatical model of the source programming language. This model is called context-free grammars and this second phase of the compiler is called syntax analysis or parsing.

Although grouping characters into words may seem similar to fitting words into a sentence, a regular expression or a finite automaton cannot recognize the context-free grammar due to the limited states available.

For example, it is impossible for a finite automaton to recognize an arbitrarily long balanced parentheses, because a machine with N states cannot remember a parenthesis-nesting depth greater than N.

Another example is the recursive abbreviation. Consider the following grammar that describes an "expression":

$$digits = [0-9]+$$

$$sum = expr\text{“}+\text{”}expr$$

$$expr = \text{“(”}sum\text{“)”}|digits$$

this expression is designed for defining forms as follows:

$$(1 + (2 + 3))$$

If we substitute *sum* into *expr*, we get the following:

$$expr = \text{“(”}expr\text{“}+\text{”}expr\text{“)”}|digits$$

And if we substitute expr into itself, we get the following:

$$expr = \text{“(”}\text{“(”}expr\text{“}+\text{”}expr\text{“)”}|digits\text{“}+\text{”}expr\text{“)”}|digits$$

Obviously, the occurrences of *expr* could be any number. However, a finite state machine or finite automaton cannot require an arbitrary amount of memory. Therefore, the recursive abbreviation cannot be recognized by a regular expression.

Therefore, instead of using the finite automaton, the parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation, called parse tree, that describes the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. Figure 5.12 presents the output of the syntaz analyzer as applied to the counter example.

```
Source:  (at 1)
  Description:  (at 1)
    ModuleDef: counter (at 1)
      Paramlist:  (at 0)
      Portlist:  (at 1)
        Ioport:  (at 2)
          Output: count, False (at 2)
            Width:  (at 2)
              IntConst: 7 (at 2)
              IntConst: 0 (at 2)
          Reg: count, False (at 2)
            Width:  (at 2)
              IntConst: 7 (at 2)
              IntConst: 0 (at 2)
        Ioport:  (at 3)
          Input: enable, False (at 3)
        Ioport:  (at 4)
          Input: clk, False (at 4)
        Ioport:  (at 5)
          Input: reset, False (at 5)
      Always:  (at 8)
        SensList:  (at 8)
          Sens: posedge (at 8)
            Identifier: clk (at 8)
        Block: None (at 8)
          IfStatement:  (at 9)
            Identifier: reset (at 9)
            NonblockingSubstitution:  (at 9)
              Lvalue:  (at 9)
                Identifier: count (at 9)
              Rvalue:  (at 9)
                IntConst: 8'b0 (at 9)
            IfStatement:  (at 10)
              Identifier: enable (at 10)
              NonblockingSubstitution:  (at 10)
                Lvalue:  (at 10)
                  Identifier: count (at 10)
                Rvalue:  (at 10)
                  Plus:  (at 10)
                    Identifier: count (at 10)
                    IntConst: 1 (at 10)
```

Figure 5.12: Syntax analyzer result of processing the counter.v

Source code of the lexical analyzer and the syntax analyzer are listed in the appendix. Instead of writing a compiler front end from sketch, the open source library Python Lex Yacc (PLY) [61] and Pyverilog [62] are used to implement the scanner and the parser. Since the main objective of this chapter is to introduce the HDL converter which upgrades a Verilog design to a more reliable version rather than to design a compiler front end from sketch. The discussion here is focuses on the general mechanism behind these tools. Details of the design and implementation of the lexical analyzer and the syntax analyzer are beyond the scope of this thesis. The readers can refer to [59][58][60] for a full description.

5.3.3 Intermediate Code Modifier

After the intermediate representation of an abstract syntax tree is generated, the next phase is to modify the AST data structure.

A human being modify a code in three steps. First we read the code and try to understand the intent of each line of code. Then we locate the part that needs to be changed. Finally,we make modifications based on the requirements and language grammar.

The lexical analyzer and the syntax analyzer introduced in the previous section complete the first task and create a data structure for later usage. In order to locate the part that needs to be changed, a tree traversals function is required [60]. This tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme. As introduced in Figure 5.13, a traversal of a tree starts at the root and visits each node of the tree in some order. Figure 5.14 illustrates the pseudocode of the visit function.

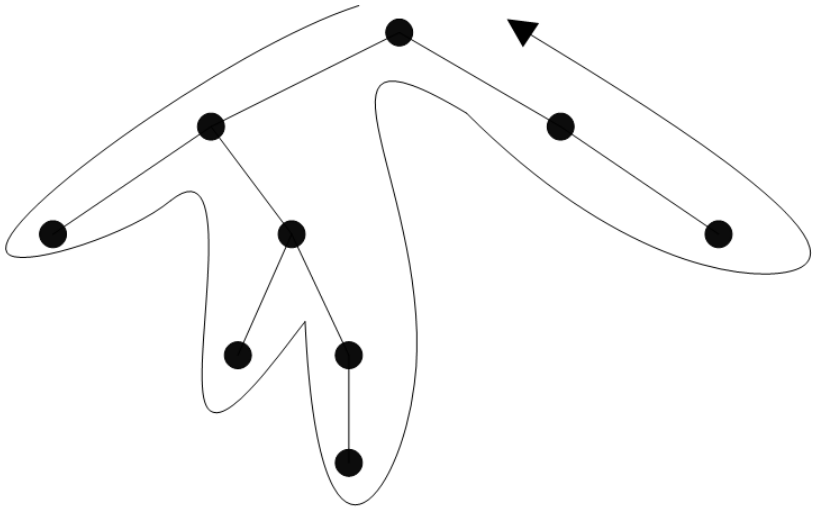

Figure 5.13: Tree Traversal Diagram.

```
procedure visit(node N) {
for ( each child C of N, from left to right ) {
visit (C);
}
Actions at node N;
}
```

Figure 5.14: Tree Traversal Pseudocode

For example, *visit_Reg*, presented in Figure 5.15, is a visit function that traverses the abstract syntax tree and searches all the nodes that represent a register. As a result, all the registers inside a module are located and stored in a hash table. This function is used to target all the internal sequential state registers. The hash table in Figure 5.16 is an example of applying the *visit_Reg* on the counter abstract syntax tree. This hash table is then passed to the *visit_Portlist* function to insert the new input/output ports as illustrated in Figure 5.17.

First, the existing ports are converted into a list and stored in the new_ports. Then we append new ports to the existing port list by creating new $Ioport$ item. The first $Ioport$ item inserted is the input port of the multiplexer control signal, here it is represented by the $self.signal$ variable which can be easily renamed. The following ports are

the state synchronization data port from the nearby modules. One is an input port starts with $neighbor\_in$, the other one is an output port starts with $neighbor\_out$. Both of their names are followed by the state register's name and the corresponding width of that register. This extended new_ports list will then be used to generate the update Verilog code using the HDL code generator introduced later.

```
    def visit_Reg(self, node):
        self.vardict[node.name] = [node.width.msb, node.width.lsb
            ]
        return self.generic_visit(node)
```

Figure 5.15: Example of creating a state register hash table. The hash table *vardict* stores the state register name and the corresponding width

```
{'count': [7, 0]}
```

Figure 5.16: The state register hash table of the counter.

```
    def visit_Portlist(self, node):
        new_ports = list(node.ports)
        new_ports.append(Ioport(Input(self.signal, width=Width(
            IntConst('0'),IntConst('0')))))
        for k, v in self.vardict.items():
            new_ports.append(Ioport(Input('neighbor_in_'+k, width
                =Width(IntConst(str(v[0])),IntConst(str(v[1]))))))
            new_ports.append(Ioport(Output('neighbor_out_'+k,
                width=Width(IntConst(str(v[0])),IntConst(str(v[1])
                )))))

        node.ports = tuple(new_ports)
        return self.generic_visit(node)
```

Figure 5.17: Example of an intermediate code modifier function. This function inserts the signal port for controlling the multiplexer, this signal is the "sync" signal passed from outside; the '$neighbor\_in\_'+k$ and the '$neighbor\_out\_'+k$ port is the state synchronization data port from/to the neighbor's module. The "k" in the name field will be replaced with the information from the state synchronization hash table.

5.3.4 HDL Code Generator

The HDL code generator generates a source code in Verilog HDL from the intermediate representation of an modified AST. Figure 5.18 shows an example python code of generating a Verilog file from the AST. In this example, an always block is generated, and the AST object is built from scratch only for demonstrating the functionality of the HDL code generator. However, when the HDL code generator is integrated to the HDL converter, it generates the Verilog code based on the upgraded syntax tree created by the intermediate code modifier. As presented in the code, each word is reconstruct into a "sentence" based on their syntactic category, and the "sentence" is organized into the source program based on the Verilog grammar. Figure 5.19 illustrates the generated source code by the Python script in Figure 5.18.

```
sens = vast.Sens(vast.Identifier('CLK'), type='posedge')
senslist = vast.SensList([ sens ])

assign_count_true = vast.NonblockingSubstitution(
    vast.Lvalue(vast.Identifier('count')),
        vast.Rvalue(vast.IntConst('0')))
if0_true = vast.Block([ assign_count_true ])

# count + 1
count_plus_1 = vast.Plus(vast.Identifier('count'), vast.IntConst(
   '1'))
assign_count_false = vast.NonblockingSubstitution(
    vast.Lvalue(vast.Identifier('count')),
    vast.Rvalue(count_plus_1))
if0_false = vast.Block([ assign_count_false ])

if0 = vast.IfStatement(vast.Identifier('RST'), if0_true,
   if0_false)
statement = vast.Block([ if0 ])

always = vast.Always(senslist, statement)
```

Figure 5.18: Example of the python source code creats an always statement using HDL code generator.

```
always @(posedge CLK) begin
  if(RST) begin
    count <= 0;
  end else begin
    count <= count + 1;
  end
end
```

Figure 5.19: Example of the generated Verilog code of an always statement.

In order to build the always block presented in Figure 5.19, we first create the sensitive list, which is the content inside the parenthesis after the @ symbol. Then we build the if statement by first developing the expression for the case when the if condition is true. This expression is essentially a non blocking substitution which is consited of a Lvalue and a Rvalue. Similarly we build the expression for the false condition. This expression is a little bit complicated since the Rvalue is no longer a simple constant. Instead, it is a PLUS expression, which is $count + 1$ in this example. Next, we combine the condition, the if_ture statement and the if_false statement to complete the if statement. Finally, we group the sensitive list and the if statement to finish the always statement.

In summary, the HDL converter consists of four parts: a lexical analyzer, a syntax analyzer, an intermediate code modifier and a HDL code generator.

The lexical analyzer reads individual characters, groups them into words, and categories each word to their parts of speech. The syntax analyzer reads distinct words and construct an abstract syntax tree based on the grammar. The lexical analyzer and the syntax analyzer composite the front end of the HDL converter, which identifies the internal sequential state registers and the input/output ports of the original design file.

The intermediate code modifier reads the parsed syntax tree and upgrades the user logic to allow state synchronization and creates top-level design for deploying self-repairing architecture. For the user logic modification, this is achieved by first adding new input ports and output ports to the current syntax tree. Second, every pin that connects to the user logic state signal are reassigned. Third, A multiplexer, a "Sync" port are added to switch

the source signal to that state register. An if statement with the synchronization condition is inserted to the original always block. Consequently, all the logic states of the recently activated module will be synchronized to the nearby module in one cycle.

For the self-repairing architecture deployment, this is achieved by first creating a top-level self-repair module with the same IO ports of the target module. Second, inside of this top-level module, multiple copies of the original module which serve as the redundant modules for the target are instantiated. Third, The enhancement module and the repair interface module are also included. Next, enhancements to detect errors and correct errors and repairing communication interface module are included. Therefore transient faults will be ignored but soft errors and hard errors will be detected using a BER counter. Upon detection of a soft fault, partial reconfiguration will be initiated.

Finally, the HDL Code Generator creates new HDL files(s) based on the modified intermediate representation.

CHAPTER 6

EXPERIMENT RESULTS

6.1 Case Study: ITC benchmark designs

In this section, we implement our enhanced-DMR/TMR/QMR approaches using standard benchmark logic designs [63] and the Xilinx 28nm Kintex-7 FPGA development platform. The studied benchmarks are listed in Table 6.1 and range in size from MSI (25-172 gate) to LSI (1,000 gate) complexity. We use these as representative of small-to-medium grain logic blocks that make up a much larger VLSI system. The specific target FPGA is the XC7K325T-2FFG900C, containing a total of 50,950 slices (203,800 LUTs, 407,600 flip flops). In the table, the first column is the ITC99 benchmark design reference [63]. The next two show the block size in gates and LUTs respectively. The fourth column lists the number of flip-flops in each block. The fifth column lists the number of primary inputs and outputs per block. The sixth column shows the number of lines of VHDL code, and the last column shows the ratio N/M where N is the number of primary outputs and M is the block size as measured in LUTs.

The benchmarks are described in VHDL and converted to the target FPGA layout using the Vivado development suite from Xilinx. Prior to layout, we replicate the VHDL code for

Table 6.1: ITC99 BENCHMARK DESIGNS

ITC99#	gates (LUTs)	FFs	I,Os	VHDL	N/M
B01	45(5)	5	4,2	110	0.4
B02	25(4)	4	3,1	70	0.25
B03	150(12)	30	6,4	141	0.333
B09	131(25)	28	3,1	103	0.04
B10	172(29)	17	13,6	167	0.207
B12	1000(207)	121	7,6	567	0.029

each block 2, 3, or 4 times to construct the enhanced DMR, enhanced TMR, or enhanced QMR versions.

6.1.1 Implementation Cost Estimation

For enhanced DMR, the bit error rate measurement logic block is shown in Figure 6.1, we use a portion of a LUT to implement the XOR function that recognizes when the two units disagree. A 2-bit counter is implemented using two available flip-flops within the same slice as the LUT. The counter is initialized (reset) by a global BER-reset signal every 2^K clock cycles, where K is the number of bits in a shared global counter. The time period between BER resets serves as the denominator in the BER ratio. The 2-bit counter is designed to signal an error upon receiving the SECOND error within that time period (the first error is ignored since it represents an acceptable BER). In the case of a soft or hard fault in either unit, the BER will reach the 2-error threshold very quickly and signal that the cell has failed and needs repair.

For enhanced TMR, we add the TMR enhanced voting logic to the tri-modular redundant configuration. Here we again use a 2-bit counter to distinguish between an acceptably-low BER from transients and the higher rates from soft or hard errors. One LUT is used to implement the voter logic and error signal. A second LUT generates a 2-bit error code that identifies the failing unit (one of three, with the fourth code used to indicate error-free operation). When the BER reaches the threshold of TWO errors within 2^K clock cycles, the system is notified and it reads the 2-bit error code in order to determine which unit to repair.

For enhanced QMR, we add the logic shown in Figure 6.2 and Figure 6.3 to the quadruple-modular redundant configuration. When implemented in FPGA, these two logic and combined together to fit into multiple LUTs. As with enhanced DMR and enhanced TMR we use a 2-bit counter to determine when the BER exceeds the threshold of TWO errors per 2^K clock cycles. However, in this arrangement the 2-bit error code is fed back into the

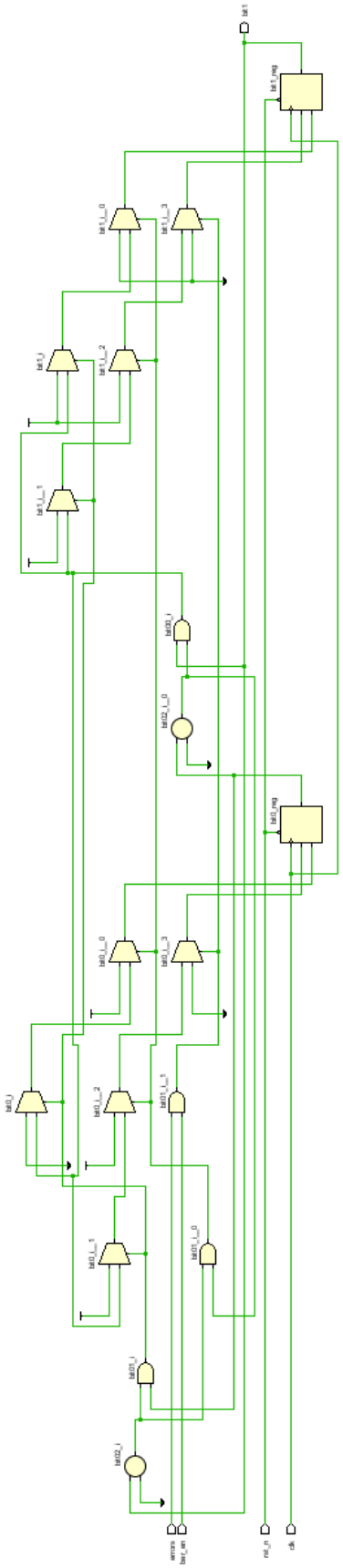

Figure 6.1: BER measurement logic block schematic

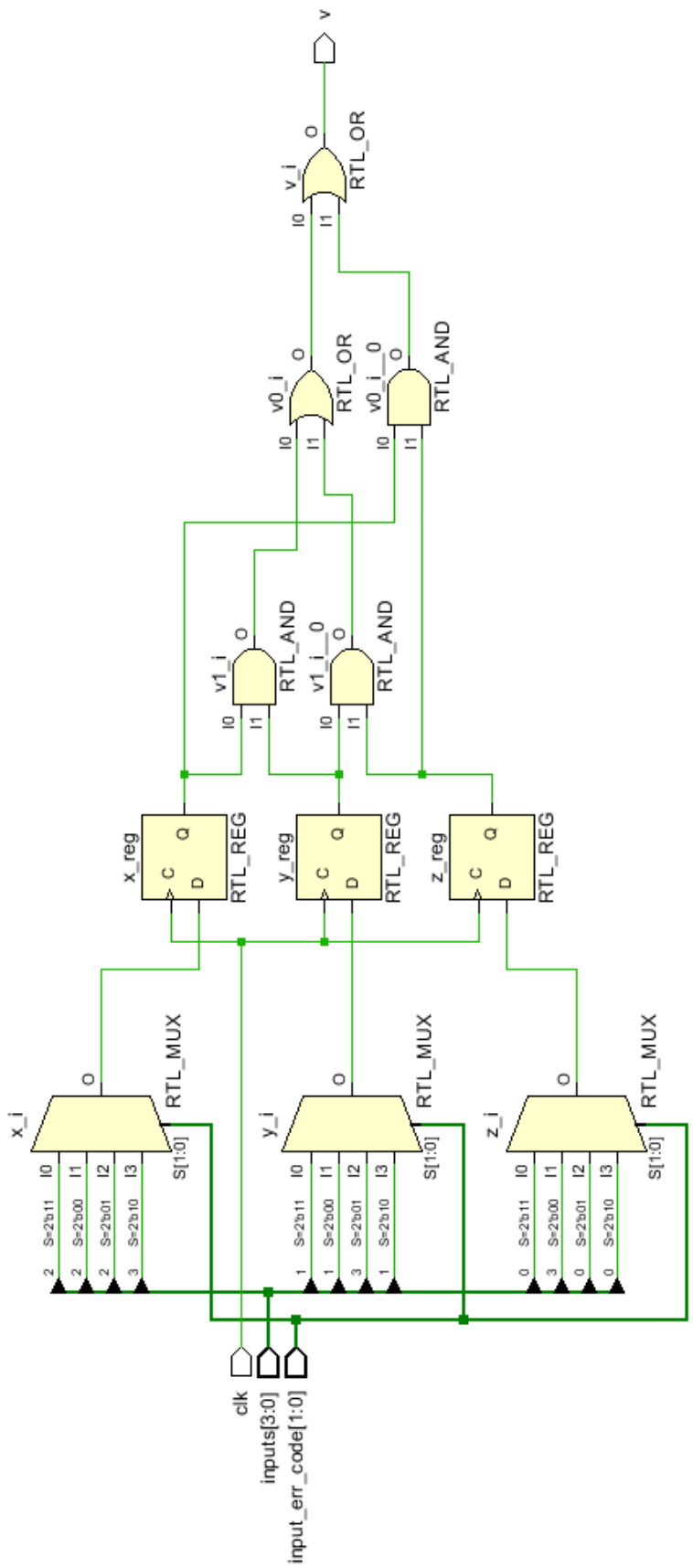

Figure 6.2: Enhanced voting logic block schematic.

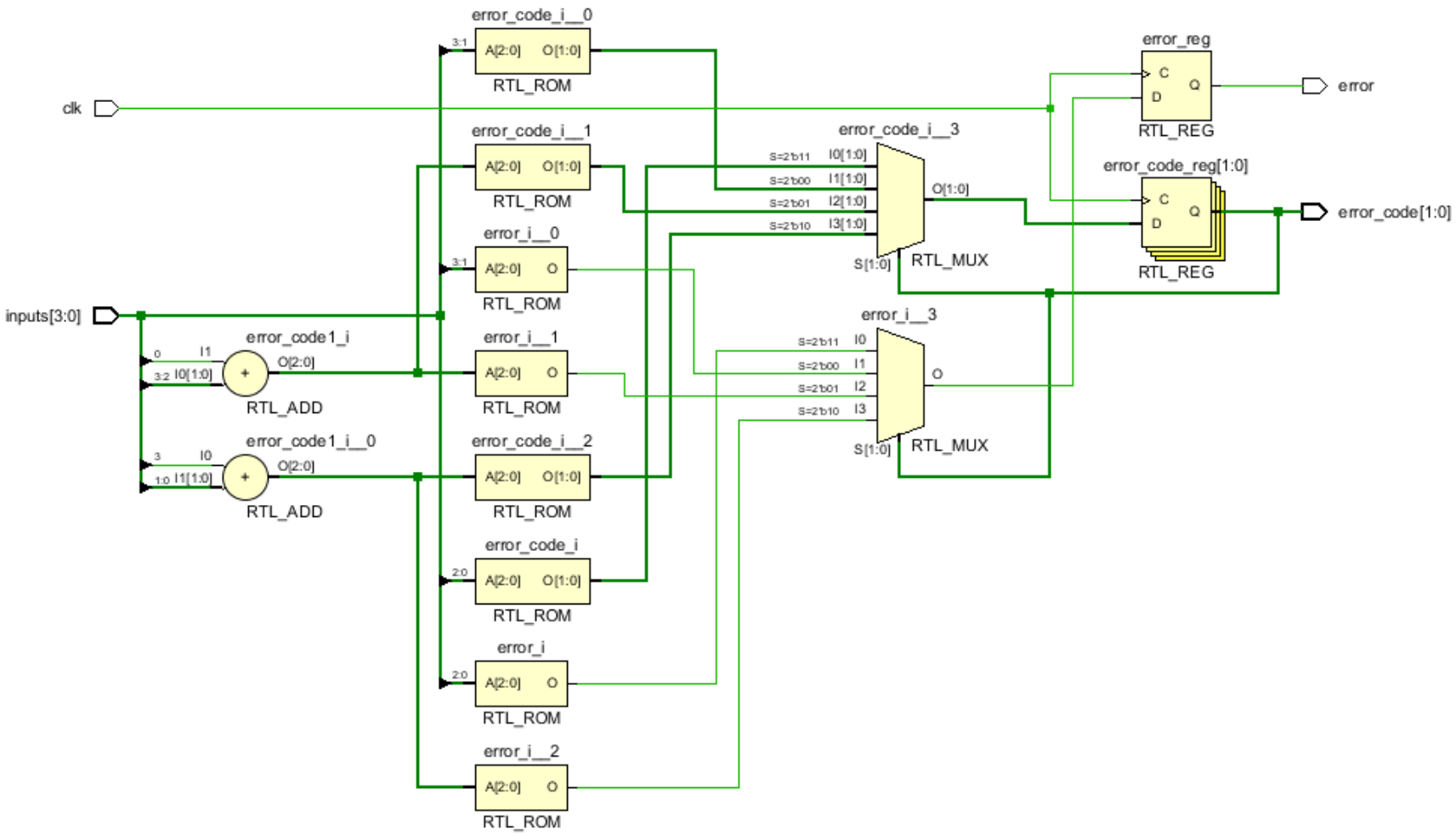

Figure 6.3: Enhanced diagnostic logic block schematic.

voter and error-generating LUTs so that the last failing unit is ignored, while allowing the remaining three units to function in TMR mode. A single soft or hard error is immediately repaired by substituting the fourth spare unit for the failing unit in the remaining enhanced TMR configuration. Again, as with enhanced TMR, if the BER ever exceeds the threshold, then the system will get a Master Error Flag signal, and it can then read the Error Code to identify the unit to repair. The failed unit is therefore isolated for re-programming to attempt healing of a soft fault.

To summarize, the added test logic requires 1, 3, or 4 LUTs for Enhanced DMR, Enhanced TMR, Enhanced QMR respectively. Therefore, the total size (in terms of LUTs) of the fault-tolerant logic blocks and the overhead costs can be approximated as follows:

$$\text{Enhanced DMR Area} = 2M + N \tag{6.1}$$

$$\text{Enhanced DMR Overhead} = \frac{2M + N}{M} - 1 = 1 + \frac{N}{M} \tag{6.2}$$

$$\text{Enhanced TMR Area} = 3M + 3N \tag{6.3}$$

$$\text{Enhanced TMR Overhead} = \frac{3M + 3N}{M} - 1 = 2 + 3\frac{N}{M} \tag{6.4}$$

$$\text{Enhanced QMR Area} = 4M + 4N \tag{6.5}$$

$$\text{Enhanced QMR Overhead} = \frac{4M + 4N}{M} - 1 = 3 + 4\frac{N}{M} \tag{6.6}$$

Where M is the number of LUTs in the original function and N is the number of primary outputs from the logic block. In the formulas the overhead values are expressed in terms

of the original function size. For example, an overhead of 2.15 corresponds to 215% overhead, or a little more than twice the original block size. The total area in that case is 3.15 times the original block size. The formulas for Enhanced DMR, Enhanced TMR, and Enhanced QMR overhead are plotted as a function of block size in Figure 6.4, Figure 6.5 and Figure 6.6. In each case, the overhead asymptotically approaches the limits determined by the degree of redundancy, namely 100%, 200%, and 300% for Enhanced DMR, Enhanced TMR, Enhanced QMR respectively as N/M approaches zero.

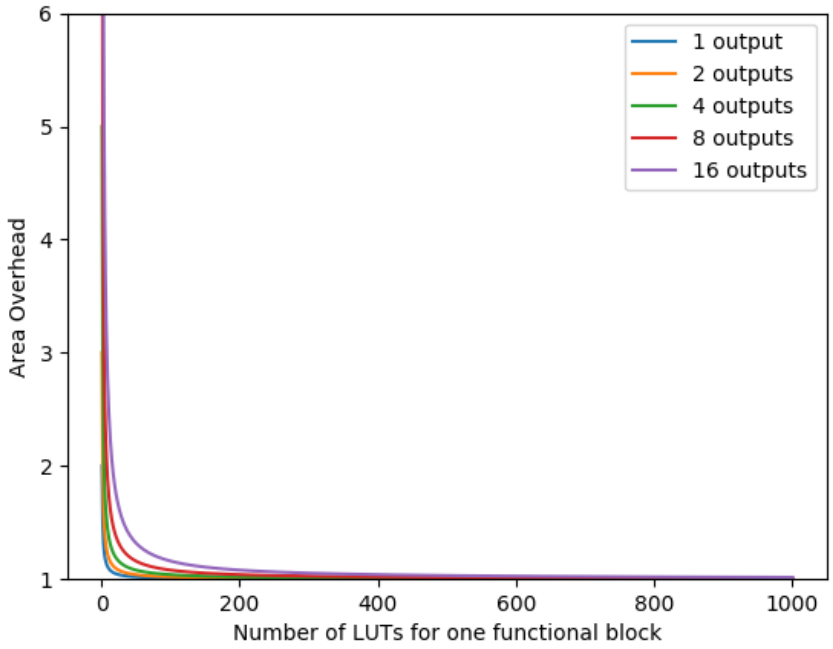

Figure 6.4: Enhanced DMR area overhead vs. the function module size.

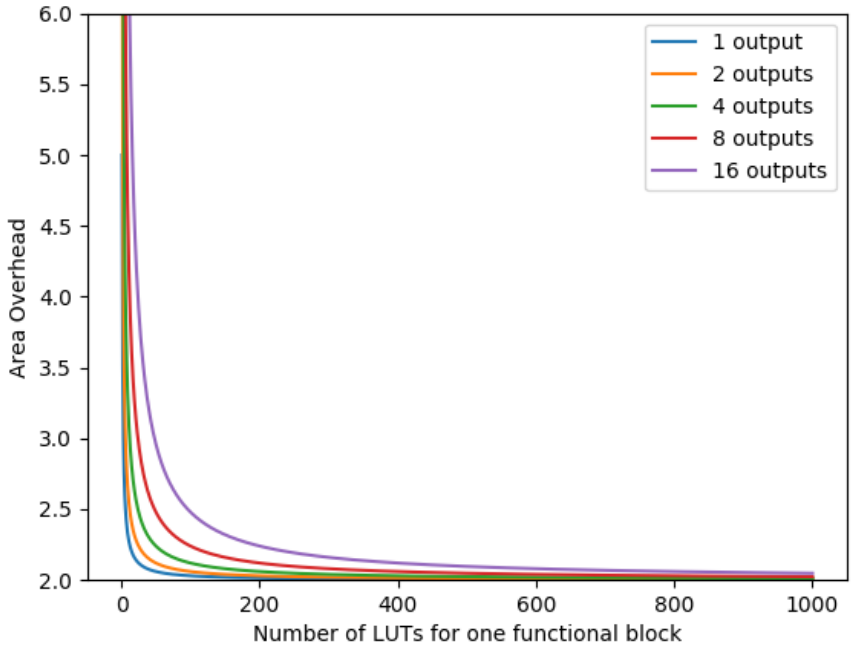

Figure 6.5: Enhanced TMR area overhead vs. the function module size.

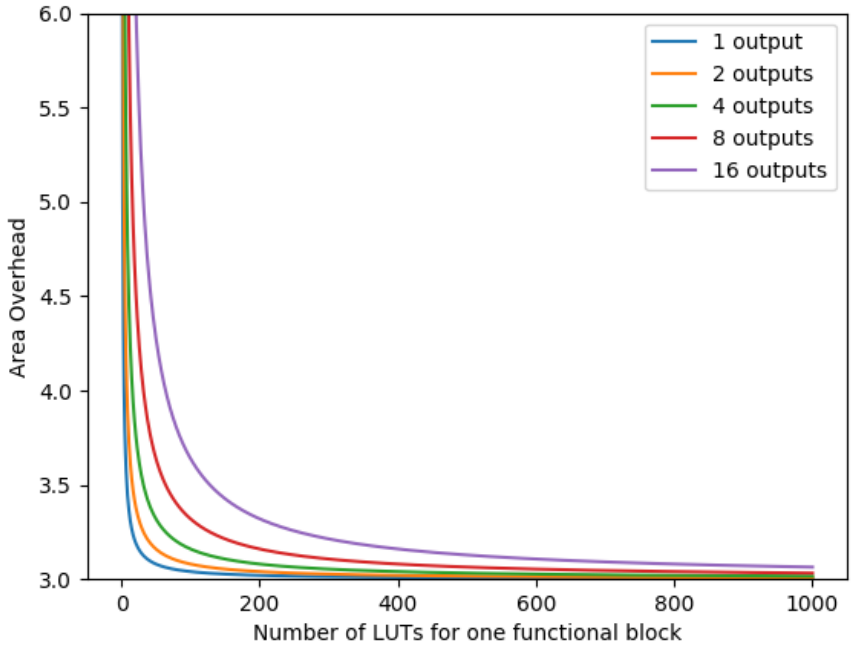

Figure 6.6: Enhanced QMR area overhead vs. the function module size.

The QMR enhancement logic block is presented in Figure 6.7. Since these enhanced modules are required for each output of the user logic block, it is best to partition the logic into large functional blocks (>100 LUTs) with few (<10) primary outputs, if possible. Also notice that these formulas are approximate in that they do not completely reflect all the design constraints for routing in an FPGA platform.

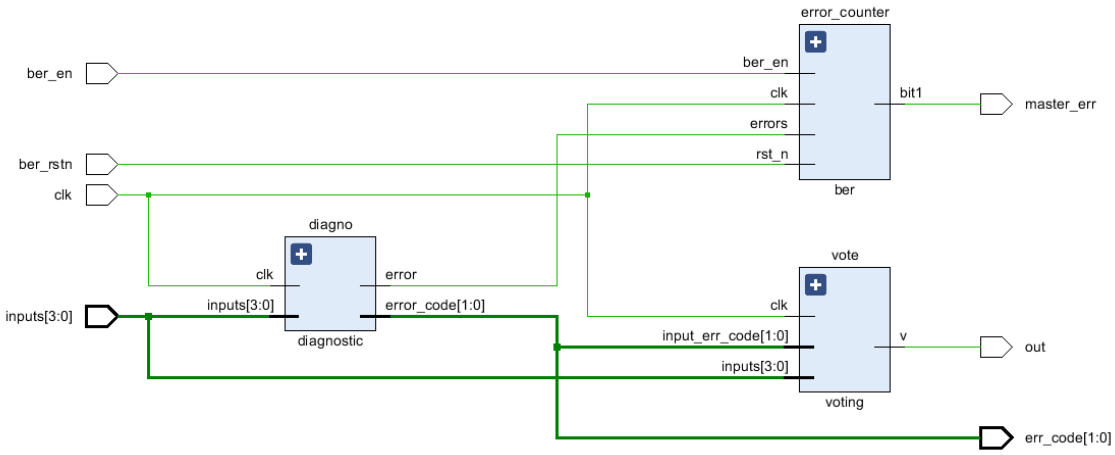

Figure 6.7: QMR enhancement logic block schematic.

To see how accurate these approximations are, we used the Xilinx Vivado design tools to layout enhanced DMR, enhanced TMR, and enhanced QMR versions of the six ITC99 benchmarks. The actual number of LUTs used in each design is shown in Table 6.2. The estimated sizes are shown in Table 6.3 for comparison. The difference between the actual layout areas and estimated results is due to the Vivado synthesis optimization.

Table 6.2: ITC99 benchmark designs implementation sizes from layout

Benchmark	**Origin**	**Enhanced DMR**	**Enhanced TMR**	**Enhanced QMR**
b01	5	11	17	27
b02	4	9	13	22
b03	12	32	47	82
b09	25	49	73	101
b10	29	78	108	176
b12	207	449	659	939

Table 6.3: ITC99 benchmark designs implementation sizes from formulas

Benchmark	Origin	Enhanced DMR	Enhanced TMR	Enhanced QMR
b01	5	12	21	28
b02	4	9	15	20
b03	12	28	48	64
b09	25	51	78	104
b10	29	64	105	140
b12	207	420	639	852

In addition, we wanted to make sure that the fault-tolerant designs did not significantly impact the overall system performance (i.e. maximum frequency). Therefore, for all the Vivado layouts we specified reasonable timing constraints. As a result, we were able to get fault-tolerant designs with nearly the same performance as the originals. This is presented in Table 6.4 and Figure 6.8, where the maximum operating frequencies are given for these same layouts. The typical loss of performance is 1-2%, with a maximum loss of 6% for the most complex design, b12.

Table 6.4: ITC99 benchmark designs maximum operating frequencies (in MHz)

Benchmark	Origin	Enhanced DMR	Enhanced TMR	Enhanced QMR
b01	108.613	108.026	103.961	107.411
b02	109.565	105.02	105.363	108.483
b03	106.112	99.226	99.088	103.659
b09	109.565	104.91	104.778	109.493
b10	104.373	104.221	99.522	102.062
b12	108.483	99.334	97.809	101.368

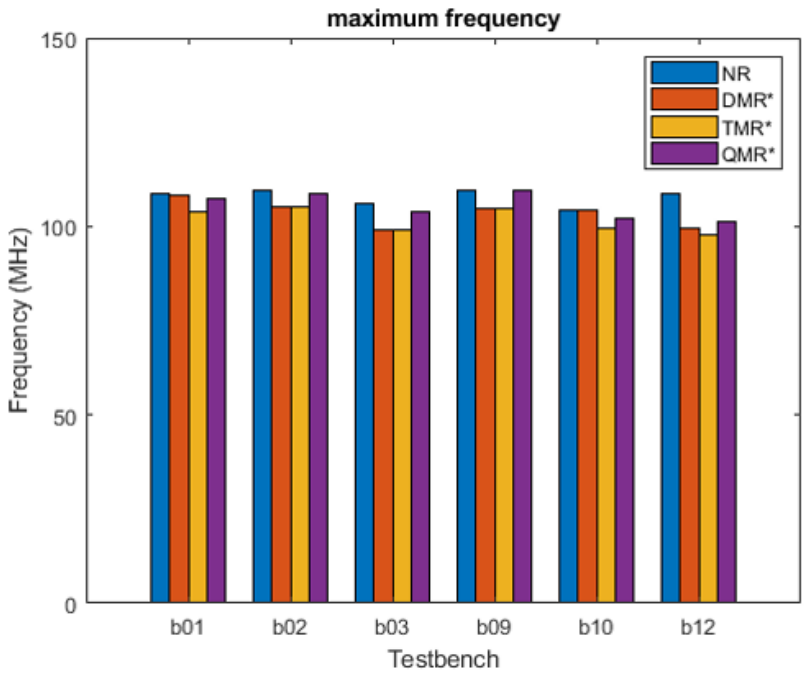

Figure 6.8: Maximum operating frequencies for ITC99 benchmark designs

Figure 6.9 shows an example layout floorplan (generated by Vivado) for the enhanced QMR version of b12. This drawing represents only about 0.5% of the entire Xilinx FPGA, so up to 200 of these LSI cells could fit on the chip. The four copies of the original b12 function are shown as clusters of red-colored logic slices. Each cluster is about 207 LUTs, but varies slightly across the four versions, presumably as a result of small optimizations made by Vivado to meet other design constraints. The small group of blue-colored squares represents the logic slices used to implement the voter and other test logic for all six primary outputs.

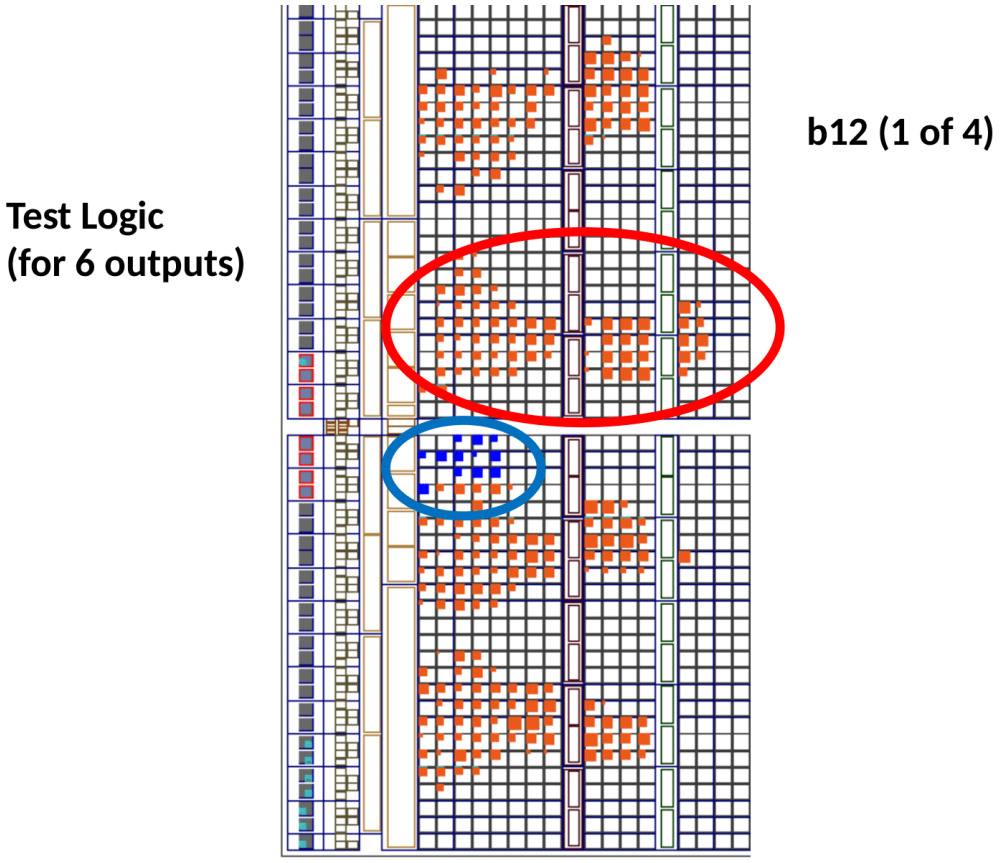

Figure 6.9: Layout floorplan for enhanced QMR version of b12

6.1.2 Reliability Model

Combinatorial and Markov models are the two fundamental approaches to modeling hardware reliability [64].

Combinatorial models use probabilistic techniques to enumerate the ways in which a system can remain operational. The reliability of a system is generally derived in terms of the reliabilities of each individual component [65]. Although this model is effective for evaluating simple systems, complex systems, which involves fault recovery and module repairing, are often difficult to be incorporated. [64].

Markov models can overcome the recovery problems. A Markov chain, or more precisely a first-order Markov chain, is a stochastic process whose dynamic behavior is such

that probability distributions for its future development depend only on the present state and not on how the process arrived in that state [66].

Figure 6.10 presents an example of modeling the reliability of a computing system with a two-state Markov chain. State 0 represents the system is functional and State 1 means the

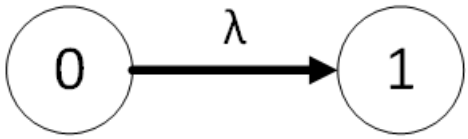

Figure 6.10: Markov model for a two state system

system is failed. The system transitions from functional to failed at a rate of λ , as labeled on the arc from State 0 to State 1. In addition, we denote by $P_j(t)$ the probability of the system being in State j at time t. Therefore the incremental change dP_0 in probability of State 0 at increment of time dt can be expressed as:

$$dP_0 = -(P_0)(\lambda dt) \tag{6.7}$$

$$\frac{dP_0}{dt} = -\lambda P_0 \tag{6.8}$$

Moreover, since the device is known to be healthy at initial time t = 0, with the initial condition $P_0(0) = 1$, the solution for this differential equation is presented as follows:

$$P_0(t) = e^{-\lambda t} \tag{6.9}$$

Since only State 0 is the functional state, the system reliability R is:

$$R(t) = P_0(t) = e^{-\lambda t} \tag{6.10}$$

This result can be verified by [4].

Xilinx provides a detailed study that describes the measured reliability characteristics of its FPGA technology [43]. For the 50,000 LUT, 28nm Kintex-7 chip, the relevant reliability

numbers are:

- Transient failure rate (λ_t) = 73728 FIT
- Soft failure rate (λ_s) = 7373 FIT
- Hard failure rate (λ_h)= 11 FIT

The FIT stands for Failures In Time. In this report, the number represents the number of errors occurred per billion hours. Therefore, the failure rate in hours should be

$$\lambda = \frac{\text{FIT}}{10^9} \tag{6.11}$$

For a non-redundant design, any type of fault will lead to a system failure. Therefore, the reliability of a non-redundant system is calculated as follows:

$$R(t) = P_0(t) = e^{-(\lambda_t+\lambda_s+\lambda_h)t} \tag{6.12}$$

Based on the study in [67][68], we present the Markov model of the transitional TMR system. As illustrated in Figure 6.11, State 3 represents a state when all three modules are functional. State 2_t means one of the three modules has a transient fault. Similarly, State 2_p stands for a single permanent fault (soft fault or hard fault). State F means more than one error occur thus the system is failed. λ_p is the permanent failure rate and $\lambda_p = \lambda_s + \lambda_h$. Since single transient error do not collapse a TMR system, thus the transient error recovery rate u is included. The TMR system reliability is calculated as follows:

$$\frac{dP_3}{dt} = -3\lambda_t P_3 - 3\lambda_p P_3 + uP_{2t} \tag{6.13}$$

$$\frac{dP_{2t}}{dt} = 3\lambda_t P_3 - \lambda_p P_{2t} - 2(\lambda_p + \lambda_t)P_{2t} - uP_{2t} \tag{6.14}$$

$$\frac{dP_{2p}}{dt} = 3\lambda_p P_3 + \lambda_p P_{2t} - 2(\lambda_p + \lambda_t)P_{2p} \tag{6.15}$$

$$R(t) = P_3(t) + P_{2t}(t) + P_{2p}(t) \tag{6.16}$$

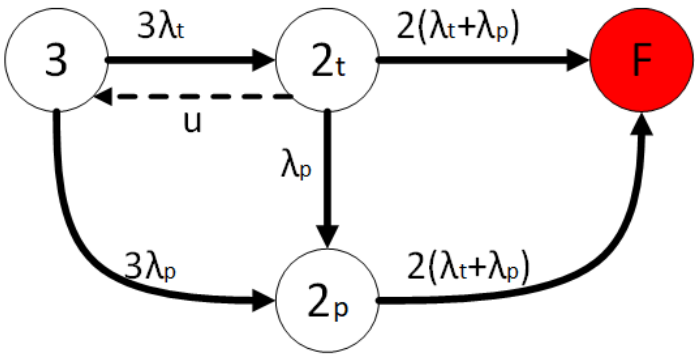

Figure 6.11: Markov model of the original TMR system.

The enhanced TMR system allows the soft faults to be identified and repaired by the healing controller. Therefore, the recovery of a single soft error must be included to the Markov model. Based on the research in [69], the Markov model of the enhanced TMR system is presented in in Figure 6.12. v is the soft error repair rate. In addition, the state that represents one permanent fault is replaced with two distinct states. State 2_s and State 2_h represent a single soft fault and a single hard fault scenario. The enhanced TMR system reliability is calculated as follows:

$$\frac{dP_3}{dt} = -3\lambda_t P_3 - 3\lambda_s P_3 - 3\lambda_h P_3 + uP_{2t} + vP_{2s} \tag{6.17}$$

$$\frac{dP_{2t}}{dt} = 3\lambda_t P_3 - \lambda_h P_{2t} - uP_{2t} - 2\lambda_a P_{2t} \tag{6.18}$$

$$\frac{dP_{2s}}{dt} = 3\lambda_p P_3 - vP_{2s} - \lambda_h P_{2s} - 2\lambda_a P_{2s} \tag{6.19}$$

$$\frac{dP_{2h}}{dt} = 3\lambda_p P_3 + \lambda_h P_{2t} + \lambda_h P_{2s} - 2\lambda_a P_{2h} \tag{6.20}$$

$$R(t) = P_3(t) + P_{2t}(t) + P_{2s}(t) + P_{2h}(t) \tag{6.21}$$

where λ_a is the sum of all types of failure rates ($\lambda_a = \lambda_t + \lambda_s + \lambda_h$).

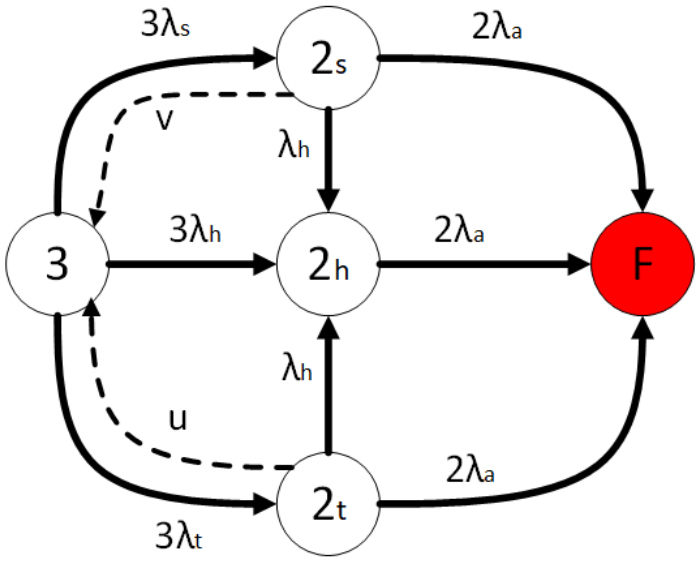

Figure 6.12: Markov model of the enhanced TMR system.

The Markov model of the enhanced QMR system based on studies in [70][64] is presented in Figure 6.13. The i and j in State $P_{i,j}$ stands for the number of the healthy working modules and the error-free spare module. For example, State $3,1$ means all three functional blocks and the back-up module are free of faults. State $3,0h$ means there is one hard fault in the system. If the fault is in the functional block, it is automatically replaced by the spare module. State $2,0ht$ means there is one hard error and one transient error in the system. The enhanced QMR system reliability is calculated as follows:

$$\frac{dP_{3,1}}{dt} = -4\lambda_t P_{3,1} - 4\lambda_s P_{3,1} - 3\lambda_h P_{3,1} + uP_{3,0t} + vP_{3,0s} \tag{6.22}$$

$$\frac{dP_{3,0s}}{dt} = 4\lambda_t P_{3,1} - 3\lambda_s P_{3,0s} + vP_{2,0ss} - vP_{3,0s} \tag{6.23}$$

$$\frac{dP_{3,0t}}{dt} = 4\lambda_t P_{3,1} - 3\lambda_t P_{3,0t} + uP_{2,0tt} - uP_{3,0t} \tag{6.24}$$

$$\frac{dP_{3,0h}}{dt} = 4\lambda_t P_{3,1} - 3\lambda_h P_{3,0h} - 3\lambda_s P_{3,0h} - 3\lambda_t P_{3,0h} + vP_{2,0hs} + uP_{2,0ht} \tag{6.25}$$

$$\frac{dP_{2,0ss}}{dt} = 3\lambda_s P_{3,0s} - vP_{2,0ss} - \lambda_h P_{2,0ss} - 2\lambda_a P_{2,0ss} \tag{6.26}$$

$$\frac{dP_{2,0tt}}{dt} = 3\lambda_t P_{3,0t} - uP_{2,0tt} - \lambda_h P_{2,0tt} - 2\lambda_a P_{2,0tt} \tag{6.27}$$

$$\frac{dP_{2,0hs}}{dt} = 3\lambda_s P_{3,0h} + \lambda_h P_{2,0ss} - vP_{2,0hs} - \lambda_h P_{2,0hs} - 2\lambda_a P_{2,0hs} \tag{6.28}$$

$$\frac{dP_{2,0ht}}{dt} = 3\lambda_t P_{3,0h} + \lambda_h P_{2,0tt} - vP_{2,0ht} - \lambda_h P_{2,0ht} - 2\lambda_a P_{2,0ht} \tag{6.29}$$

$$\frac{dP_{2,0hh}}{dt} = 3\lambda_t P_{3,0h} + \lambda_h P_{2,0hs} + \lambda_h P_{2,0ht} - 2\lambda_a P_{2,0hh} \tag{6.30}$$

$$\begin{aligned} R(t) = P_{3,1}(t) + P_{3,0t}(t) + P_{3,0s}(t) + P_{3,0h}(t) + \\ P_{2,0ss}(t) + P_{2,0hs}(t) + P_{2,0hh}(t) + P_{2,0ht}(t) + P_{2,0tt}(t) \end{aligned} \tag{6.31}$$

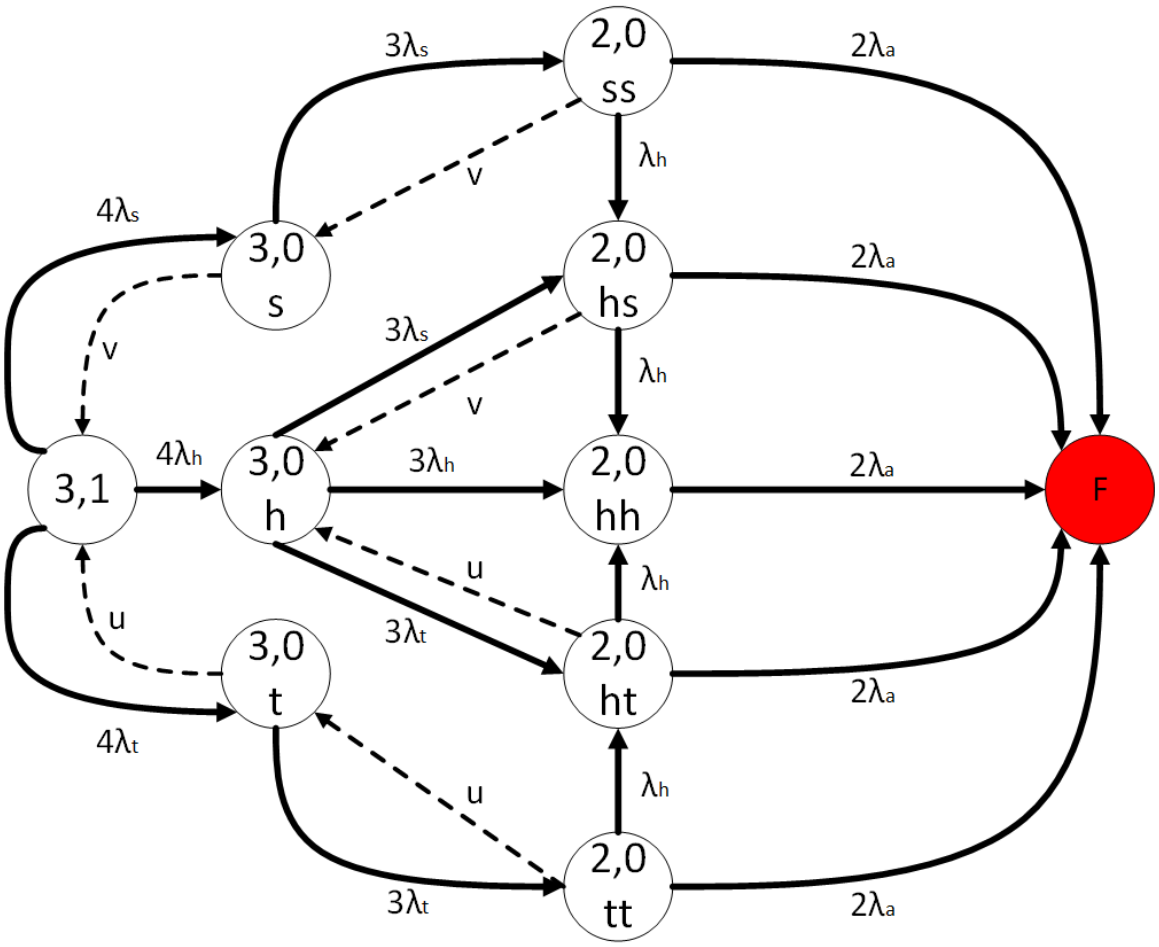

Figure 6.13: Markov model of the enhanced QMR system.

Based on the failure rate provided by Xilinx and these formulas presented above, we calculated the expected reliability as a function of time (years of operation) for a 204k LUT system (the FPGA chip), for different levels of fault-tolerance. These are plotted in Figure 6.14 and Figure 6.15 for NR, TMR, enhanced TMR, and enhanced QMR.

Because the non-redundant (NR) design is vulnerable to transient errors, its reliability function is dominated by the transient failure rate. It shows significant degradation within a year. However, TMR corrects almost all transient errors, and so its reliability is dominated by the soft failure rate. enhanced QMR self-corrects most soft and hard errors, and fails mostly from multiple hard errors within a single cell. The time scale for significant degradation of enhanced QMR is therefore measured in centuries.

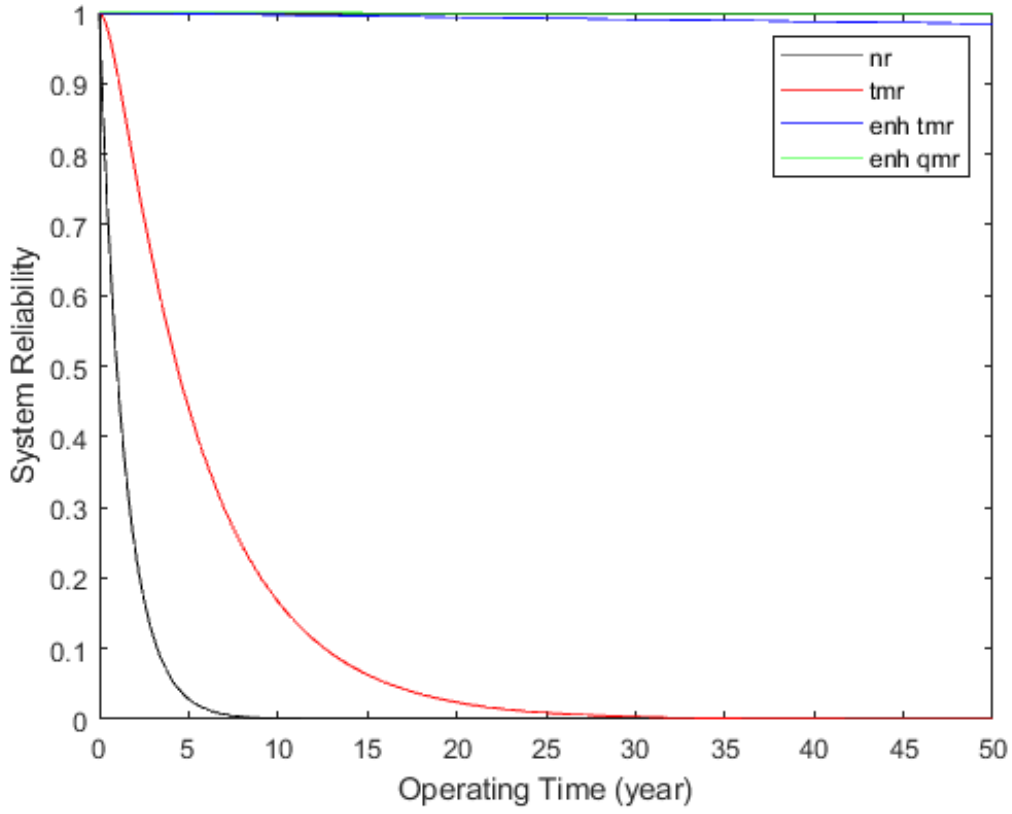

Figure 6.14: The reliability changes of a 204k LUT system in 50 years of operation.

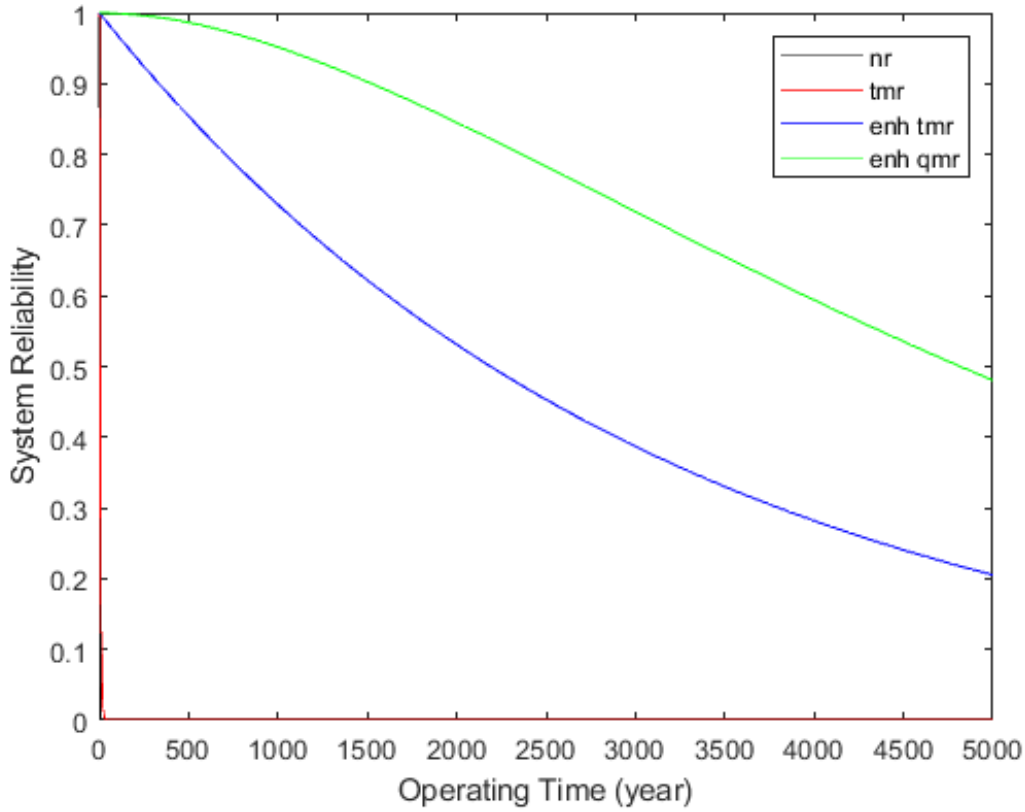

Figure 6.15: The reliability changes of a 204k LUT system in 5000 years of operation.

6.2 VLSI Case Study: Handwritten Digit Recognition

New trends in machine learning algorithm, such as using compact data types (e.g., 1-2 bit) in deep neural network, provide a great opportunity for FPGA application over traditional GPU based implementations [71]. Moreover, FPGAs outperform GPUs and CPUs in latency-sensitive applications [72]. In this section we implement our proposed self-repairing architecture using an artificial neural network (ANN) based handwritten digit classification design and the Xilinx KC705 Evaluation Board with 28nm XC7K325T-2FFG900C FPGA.

6.2.1 Experiment System Setup

As presented in Figure 6.16, the top level design contains two UART (Universal Asynchronous Receiver/Transmitter) modules, two FIFO (first in first out) modules and the neural network module. The UART RX module receives the image data stream from PC and

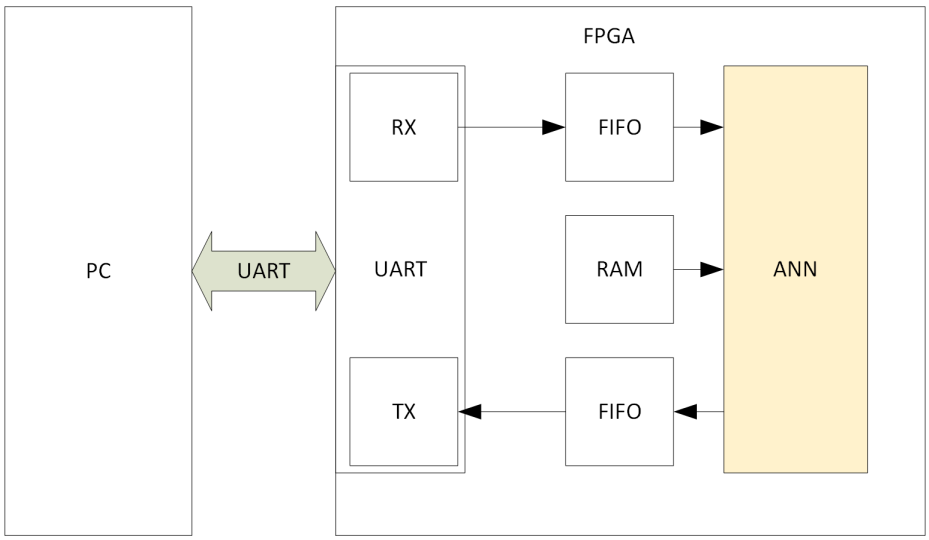

Figure 6.16: Experiment setup system overview.

the UART TX module transmits the predicted result back to PC. Since the UART protocol transforms the data between serial to parallel interface and the sampling frequency of the communication should be at least two times faster than that of the data stream. the UART

module and the ANN module work in two different clock domains. Therefore two asynchronous FIFO modules are included to eliminate the clock domain crossing issue. The trained weights are stored in the block RAM during the initial FPGA configuration.

6.2.2 ANN

An artificial neural network is a computational model based on the structure and behavior of biological neural networks, wherein each neuron sums the weighted inputs from the preceding neurons. In this application, it takes a 28x28 image and generates a 4 bit binary number representing the predicted digit. The architecture of our handwritten digit recognition neural network, is presented in Figure 6.17. The first layer is the input layer where

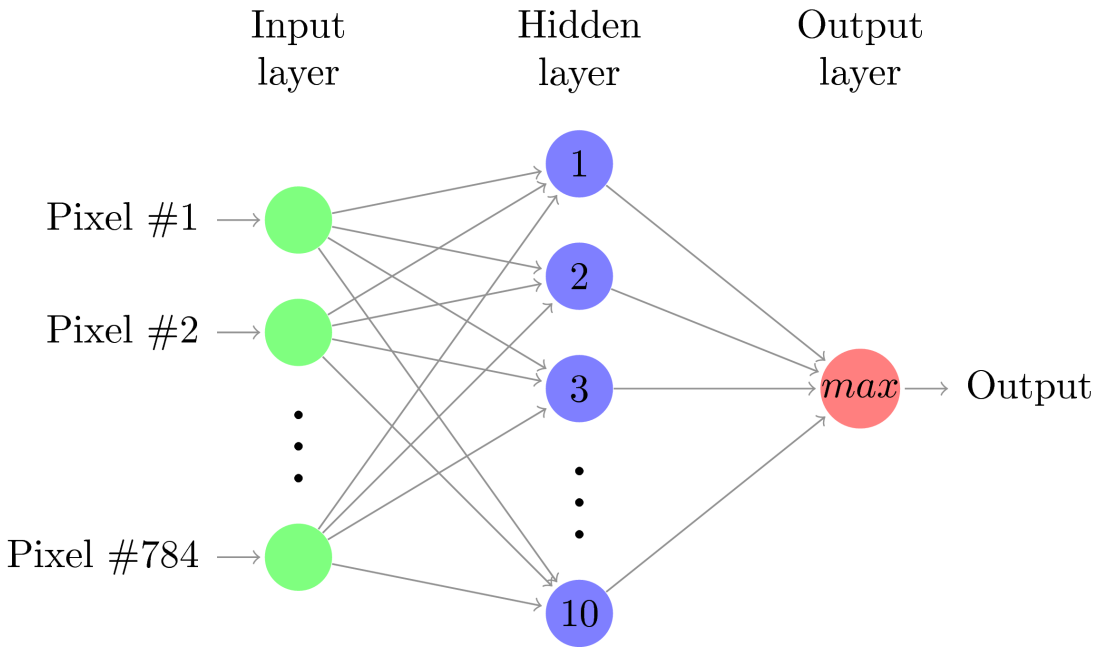

Figure 6.17: ANN architecture.

each node simply passes the image pixel to the second layer. The second layer is a layer of neurons where each neuron multiplies the pixel matrix and the corresponding weight matrix.

The general neuron behavior is presented in Figure 6.18. Notice that since our neural network model only contains a single hidden layer, the activation function is unnecessary and not implemented in this design. The output of the neuron represents the probability of the given image to be a specific digit. In our implementation, ten neurons are used representing the ten digits from zero to nine. The last layer is the output layer. It searches the index for the maximum probability from the preceding neurons. In other word it selects the digit with the highest probability.

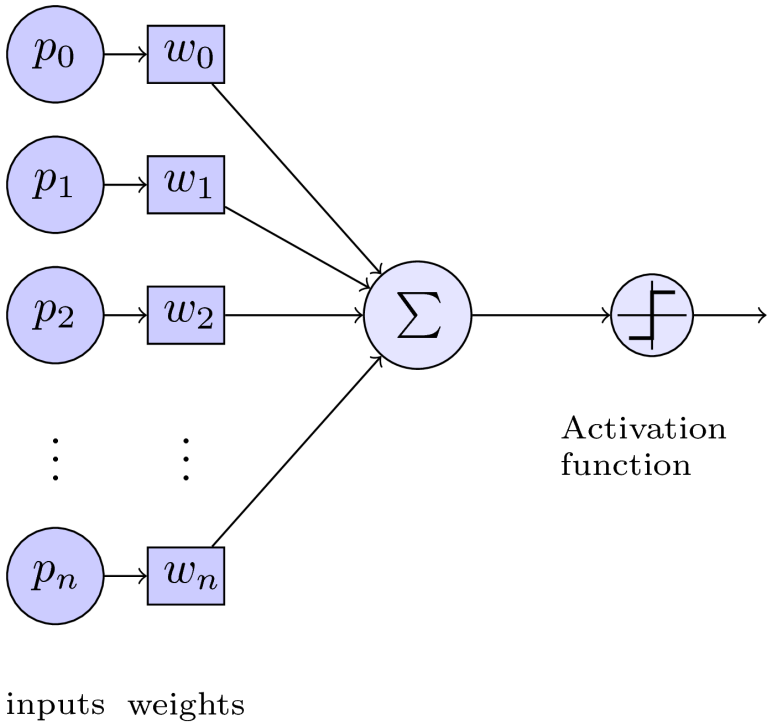

Figure 6.18: Neuron model.

The MNIST database (Modified National Institute of Standards and Technology database) of handwritten digits [73] is used to train and test our neural network model. Figure 6.19 presents an example of the handwritten digit.

The weights are generated from the software during the training process using 60,000 training image examples. These weights are pre-stored in FPGA block RAM while the handwritten digit images are streamed from PC to FPGA at run time. We use 10,000 image

examples from the MNIST testing set to evaluate the neural network accuracy under various error rates and demonstrate the robustness of our self-repairing architecture.

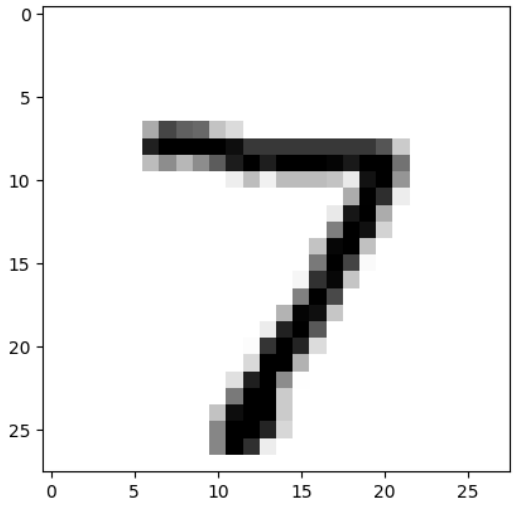

Figure 6.19: MNIST data example.

6.2.3 Error Injection

Random errors are injected to test the robustness of the self-repairing system. Since the original design is primarily consisted of a hidden layer and a output layer, two error generators are applied, as illustrated in Figure 6.20. One error generator is attached to the

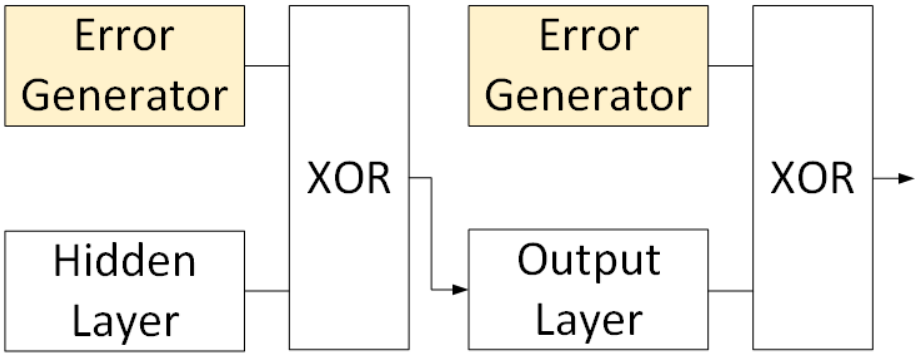

Figure 6.20: Error injection data flow.

hidden layer, mimicking the faults occur inside the hidden layer. The errors and the orig-

inal hidden layer outputs are processed through an XOR gate. For each bit of the hidden layer outputs, if the corresponding error bit is a logic one, it will flip the the hidden layer output. Otherwise, the hidden layer outputs stay unchanged. Even though the errors are not directly injected to the internal logic of the hidden layer block, from the output layer point of view, the results are the same. These corrupt results are then transported to the output layer. Similarly, the second error generator is attached to the output layer, representing the internal errors of the output layer.

The error generator module diagram is introduced in Figure 6.21. The input port, la-

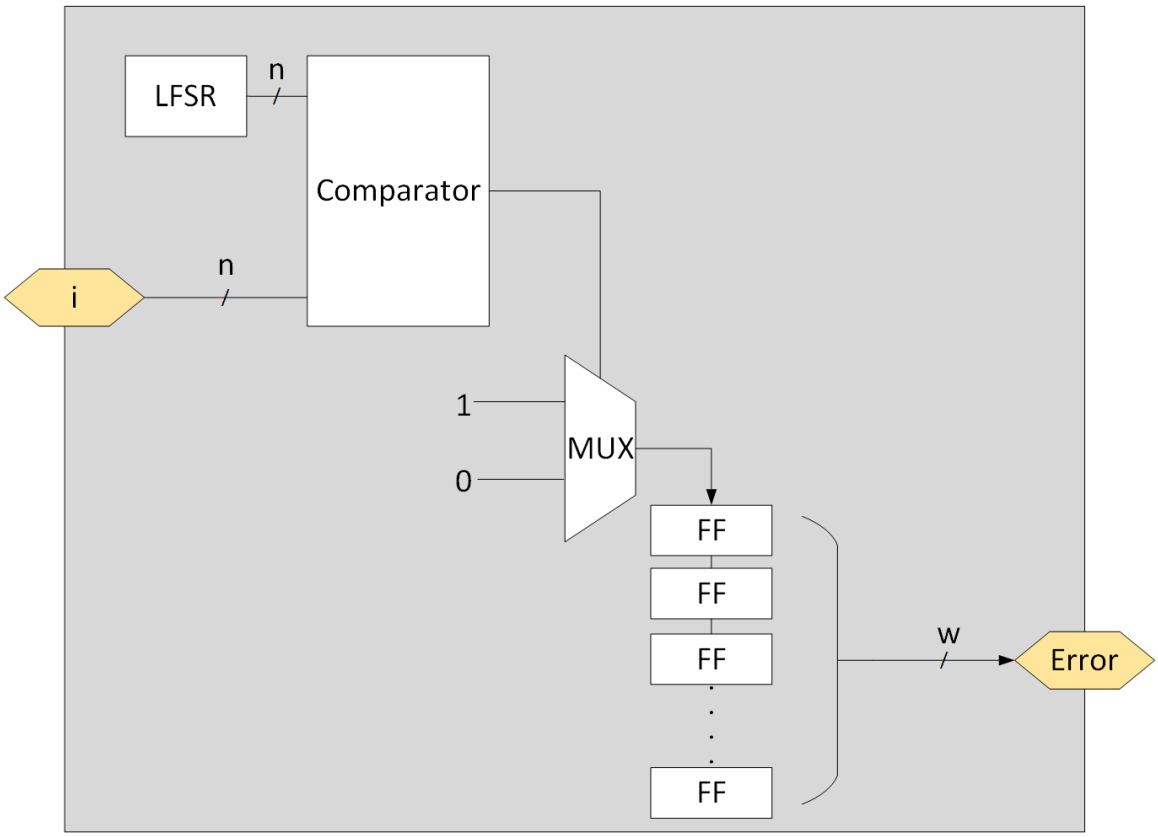

Figure 6.21: Error generator module.

beled “i” in this figure, is connected to the on-board switches to control the injected error value. If the random number generated from the Linear Feedback Shift Registers (LFSR) is less or equal to the inject error value, a logic one is generated by the comparator. Otherwise, the comparator will generate a logic zero. This output of the comparator is used to

switch the multiplexer to push an error bit (zero or one) to the filpflop chain. And this error bit will keep shifting for the following cycles. Therefore, the number of ones inside the flipflop chain depends on the probability of whether the random number generated from the LFSR is less or equal to the inject error value. For example, if the random number is evenly distributed between 1 and 10, and the inject error value is 5, then there should be 50% of ones inside the flipflop chain.

In order to obtain a random sequence with uniform probability distribution, a LFSR is used. The LFSR is a shift register whose input bit is a linear function of its previous state. Figure 6.22 presents an example of an 11-bit LFSR, the bits at position 9 and 11 perform an exclusive or operation and feed back the output to position 1. In the next cycle, the LFSR

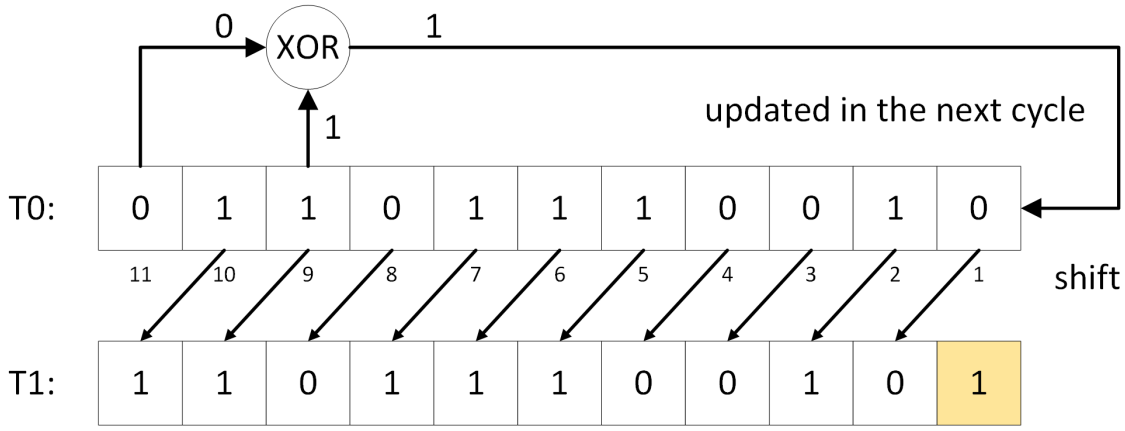

Figure 6.22: Example of a linear feedback shift register.

shifts the bits one position to the left and updates the bit at position 1 with the XOR result. The positions to perform the exclusive or operation is called a tap. For a n-bit LFSR, with well-chosen taps, this logic is capable of producing a random sequence of up to $2^n - 1$ bits without repeating itself. The mathematical theory behind this design can be found in [74].

Since the LFSR is guaranteed to generated a random sequence with constant density, the relationship between the bit error rate (probability of ones occur in the flipflop chain)

and the injected error value can be defined by the following formula:

$$e = \frac{i}{2^j - 1} \tag{6.32}$$

where e is the error rate, i is the error value from the input pin and j is the number of bits of the LFSR. For example, if the LFSR is 8 bits, and input error number is 1, the error rate is 0.392%. The error injection module output port width is designed to be the same width as the target layer result, therefore a bitwise xor operation can be performed with the correct layer results.

Figure 6.23 illustrates the enhancement module behavior. A four bit number is injected to four modules, the most significant bit corresponds to the error in module 3 and least significant bit corresponds to the error in module 1. When the first error is injected to module 1, it is detected but not asserts the error flag since the counter is not reach the threshold, the error code indicates that the faulty module is module 1 as where the error is injected and the faulty module is replaced by the backup module (module 3). Further errors from this module are temporarily masked by the enhanced voter and would not trigger the error detection. The system output "pred" as highlighted in the figure, continuously generates the correct output. When another error is injected at module 3, the error code indicates the faulty module is module 3. It switches this faulty module with module 1 and asserts the error flag since the counter reaches two. The system output "pred" still preforms correctly.

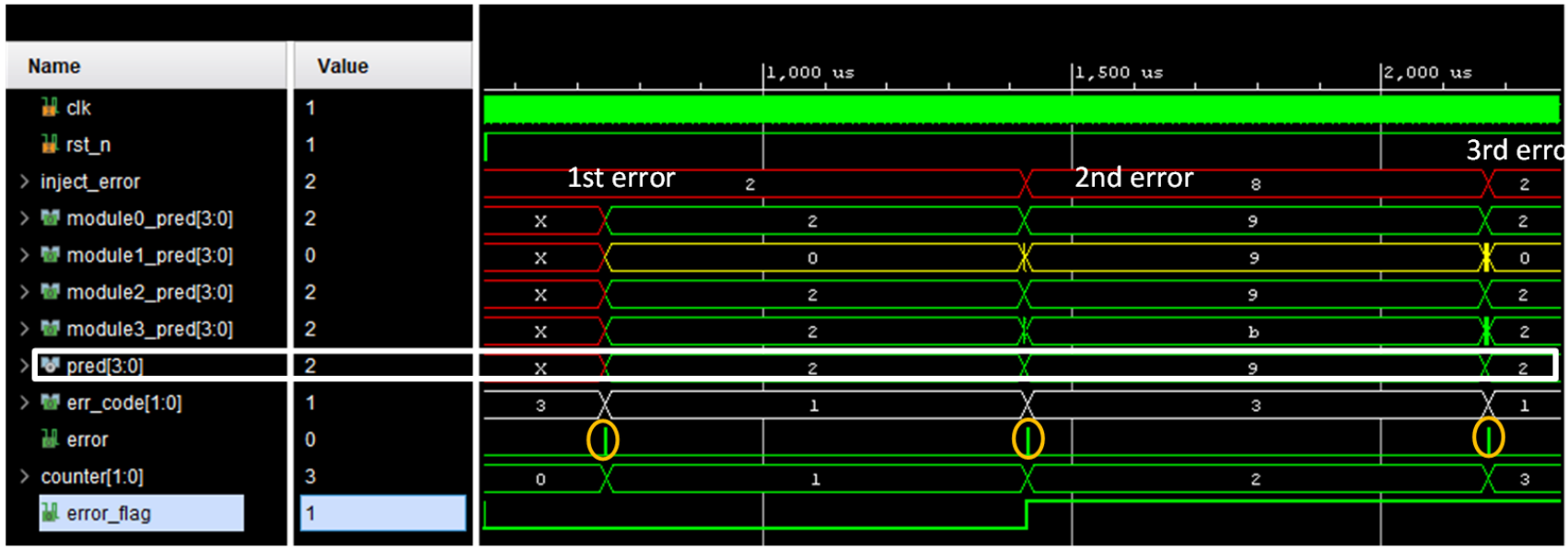

Figure 6.23: Simulation waveform of the enhancement module

6.2.4 Fault Tolerance Analysis

We use this design as our benchmark and apply our self-repairing architecture to exam the robustness of this system under error injection. The predicted results are compared with the ten thousand reference outputs from MNIST data set, summarized in Figure 6.24. Without injecting any error, the original system accuracy is 91.52%. When the error rate is relative low, the system works correctly. When the error rate increases, the original system starts

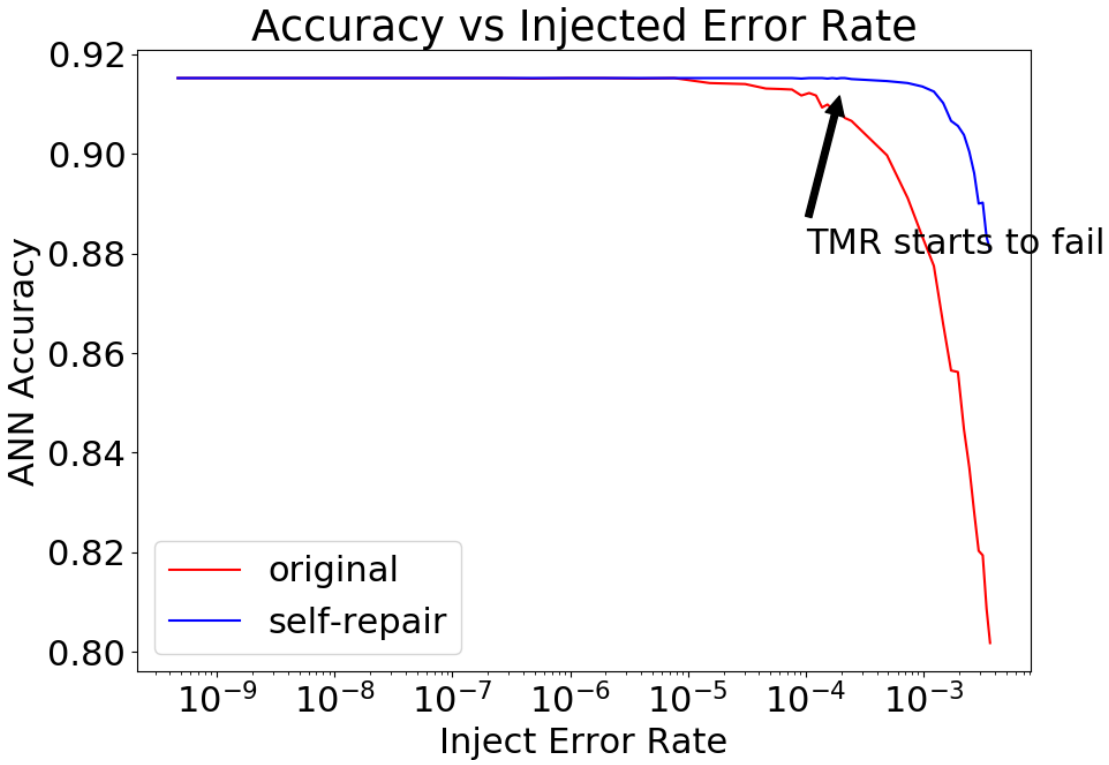

Figure 6.24: ANN accuracy vs Injected error rate.

to fail but the self-repairing system continuously produces the correct output. When the error rate elevates drastically, the accuracy of the self-repaired system starts to reduce due to to the fact that there are more than two modules affected simultaneously. The original experiment data is listed in Table 6.5 and Table 6.6.

Table 6.5: System Accuracy Results

Inject Error Rate	Original Accuracy	Self-Repair Accuracy
4.66E-10	0.9152	0.9152
9.31E-10	0.9152	0.9152
1.86E-09	0.9152	0.9152
3.73E-09	0.9152	0.9152
7.45E-09	0.9152	0.9152
1.49E-08	0.9152	0.9152
2.98E-08	0.9152	0.9152
5.96E-08	0.9152	0.9152
1.19E-07	0.9152	0.9152
2.38E-07	0.9152	0.9152
4.77E-07	0.9151	0.9152
9.54E-07	0.9152	0.9152
1.91E-06	0.9152	0.9152
3.81E-06	0.9151	0.9152
7.63E-06	0.9152	0.9152
1.53E-05	0.9142	0.9152
3.05E-05	0.914	0.9152
4.58E-05	0.9131	0.9152
6.10E-05	0.913	0.9152
7.63E-05	0.9129	0.9152
9.16E-05	0.9117	0.9151
0.000106812	0.9122	0.9152
0.00012207	0.9117	0.9152
0.000137329	0.9093	0.9152
0.000152588	0.9099	0.9151
0.000167847	0.909	0.9152
0.000183105	0.909	0.9151
0.000198364	0.9084	0.9152
0.000213623	0.9072	0.9152
0.000228882	0.9069	0.9151
0.000244141	0.9066	0.915
0.000488281	0.8997	0.9146
0.000732422	0.8911	0.9142
0.000976563	0.8833	0.9135
0.001220703	0.8775	0.9125
0.001464844	0.8658	0.9102
0.001708984	0.8565	0.9066
0.001953125	0.8562	0.9056

Table 6.6: System Accuracy Results (continued)

Inject Error Rate	Original Accuracy	Self-Repair Accuracy
0.002197266	0.8447	0.9038
0.002441406	0.837	0.9004
0.002685547	0.828	0.8961
0.002929688	0.8203	0.89
0.003173828	0.8194	0.8902
0.003417969	0.8087	0.8833
0.003662109	0.8018	0.8807

6.2.5 Implementation Report

Table 6.7 summaries the FPGA resource utilization, power consumption and timing information. The self-repairing system requires almost four times the logic resources (required for QMR) and consumes twice the power compared with the original design. The maximum frequency reduces by 10%. This QMR approach contributes the majority utilization of the system. Even though the area utilization is almost 100%, the difficulty for hardware routing does not increase significantly. As a result, the maximum frequency only drops only 10%. The blank spaces are reserved for neural network weights storage (block memory).

Table 6.7: FPGA Implementation Report

	Original	Self-Repair
LUTs (number and percentage)	39019, (19.15%)	155530, (76.32%)
Registers (number and percentage)	35770, (8.78%)	142129, (34.87%)
Slice (number and percentage)	12240, (24.02%)	48989, (96.15%)
Total On-chip Power (W)	0.334	0.505
Dynamic (W)	0.174	0.344
Static (W)	0.160	0.161
system clock frequency (MHz)	200	200
ANN clock frequency (MHz)	12.5	12.5
ANN worst negative slack (ns)	48.790	45.186
ANN maximum frequency (MHz)	32.041	28.724

The implementation layouts of the original system and the self-repairing system are displayed in Figure 6.25 and Figure 6.26. Each redundant module is presented with a unique color in Figure 6.26.

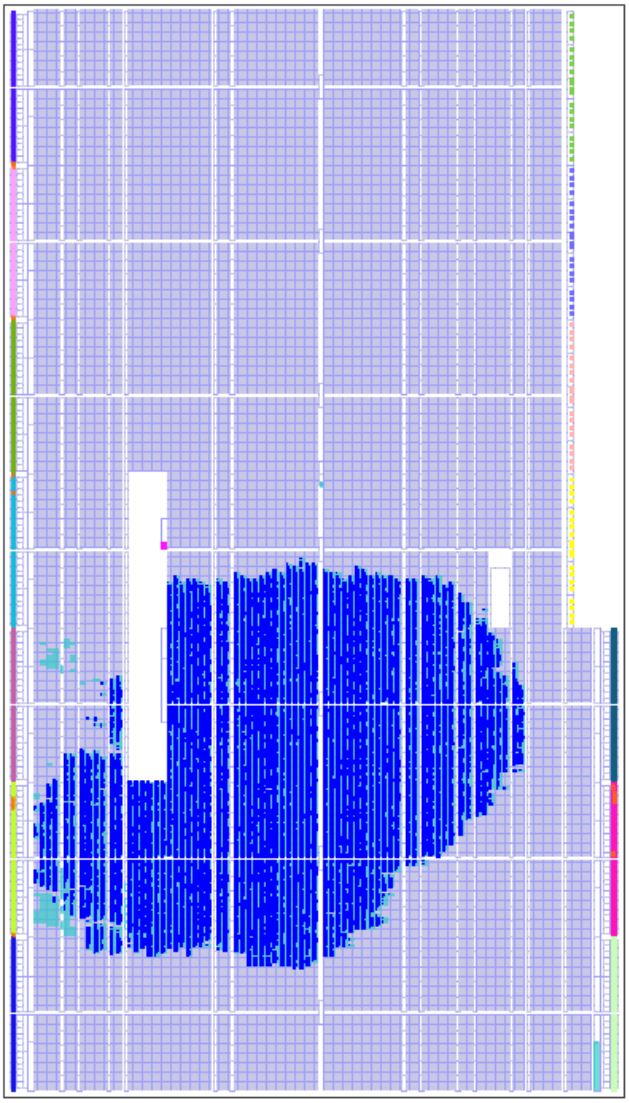

Figure 6.25: FPGA Layout of the original system.

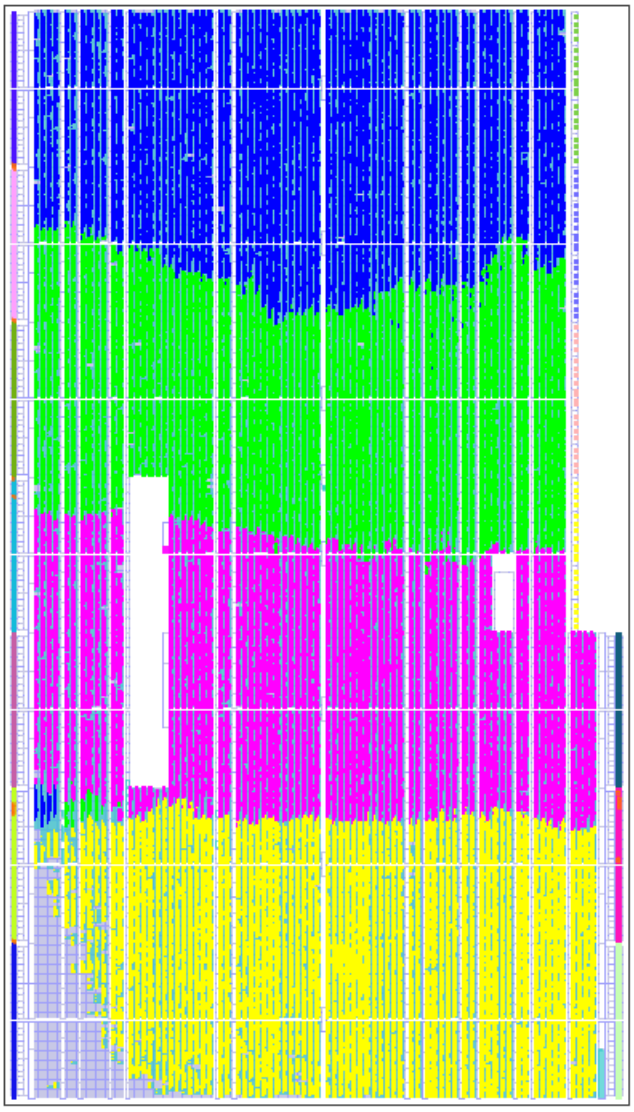

Figure 6.26: FPGA Layout of the self-repairing system. Each color represents one single neural network module.

CHAPTER 7
CONCLUSION

7.1 Summary

The objective of this research was to establish a systematic approach for the design of self-testable, self-correcting, self-repairing and self-healing digital systems. In Chapter 1, the motivation for a designing a highly reliable digital system and the purpose of this research was established.

Faults in the hardware system and their causes were reviewed in Chapter 2. This chapter also examined recent technologies to detect and repair faults, and advanced methodologies to achieve fault tolerance.

Chapter 3 presented the self-testing, self-repairing and fault-tolerant system architecture. This research mainly consisted of the development of several enhanced modules to the existing self-testing, self-repairing and fault-tolerant techniques presented in Chapter 2, aiming to improve the fault repairing efficiency, system availability and system reliability. The fault repairing efficiency improvement was achieved by using a self-testing enhancement module using a the bit error rate counter. The bit error rate counter distinguished intermittent errors from transient errors based on their effect on bit error rate: intermittent faults are more likely to occur repeatedly at the same location and occur in bursts when the fault is activated. Therefore, the transient errors are filtered and only the intermittent errors are recorded. The enhanced voting module combined with modular redundancy techniques allowed the faulty module to be identified and isolated from the healthy modules, which facilitated the repairing process. Moreover, with one extra module worked as a backup module, the system is capable of maintaining fault-tolerance even if one module is undergoing repair, which dramatically increased the system system availability and reliability.

In addition, a novel state synchronization technique that suits the self-repairing system is also presented in Chapter 3. The state synchronization technique allows the out-of-state module (due to the repairing procedure) to be synchronized with the remainder system. However, the traditional state synchronization technique only works for redundant modules of odd number. Therefore a novel state synchronization technique based on multiplexer was discussed.

Chapter 4 explained the mechanism of partial reconfiguration and the related architecture of FPGA. It also introduced the development flow of a partial reconfiguration project based on Vivado Design Suite, a Xilinx software for FPGA design, synthesis and implementation. The last section of this chapter presented the error reporting message format and the software program behind the healing controller.

In Chapter 5, A software tool, called HDL converter, that autonomously deploys the self-repairing architect and upgrades the non-safety-critical design into a more reliable version is discussed. First, the limitation of manipulating the user HDL code with the regular expression is presented. Second, the similarity and difference between the classic software compiler and the HDL converter is described. Next, the details of development of the HDL converter is presented.

Finally, various experiment results are presented in Chapter 6. First, an area cost estimation method is introduced and verified by implementing the ITC99 benchmark designs. In addition to the small area cost of the enhanced logic, we prove that the fault-tolerant designs did not significantly impact the overall system maximum frequency. Moreover, multiple Markov chain based reliability models are developed to support the evaluation theory of each enhanced self-repairing architecture. In addition, an image classification application is implemented to demonstrate the reliability improvement. In conclusion, we demonstrate the reliability improvement from both the theories and experiments, and we prove that the costs of various example design ranging from SSI (small scale integration) to VLSI (very large scale integration) are predictable and did not significantly impact the

overall system maximum frequency.

7.2 Contributions

The discussion given above highlights the works that has been done in this research. The major contributions of this thesis are illustrated below:

7.2.1 Bit Error Rate (BER) Measurement

A bit error rate enhanced testing logic is introduced in this thesis. This bit error rate counter based self-testing module detects functional errors (from changes in static logic) but ignores low BER transient errors so that higher-level repair mechanisms can be invoked at the system level only when there is a high likelihood of a structural fault. This distinction is critical for the fault repairing process since transient errors very often do not lead to a permanent functional failure. Therefore the traditional error detection method, which flags every error, is not suitable for identifying modules for repair. By filtering out transient errors from intermittent errors, the repairing procedure required by the fault management unit is effectively reduced by 90% to 95%, which significantly reduced the average single module offline time, and thus increases the system availability overall. Moreover, the hardware implementation cost of this bit error rate measurement logic is negligible. Therefore, this minimal transistors or gates requirement can largely increase the enhanced module scalability.

7.2.2 Enhancement for Fault Tolerance and Fault Isolation

An enhanced voting logic is presented in this thesis. The enhanced voting logic not only corrects the transient error generated by the faulty module, but also automatically identifies the failing unit.

7.2.3 Self-Repairing Architecture

A self-repairing architecture based on modular redundancy technique is introduced in this thesis. This architecture integrates the bit error rate measurement logic and the enhanced voting logic and achieves self-testing, fault-tolerance and immediate context switching when a permanent fault are detected.

This self-repairing architecture is easy to scale and can be applied on different hierarchy of the circuit. For example, it can be applied to the entire system, a critical subsystem, one or more modules which are more vulnerable to certain type of errors or a single data path.

7.2.4 State Synchronization

A state synchronization method is presented in this thesis. When the damaged module is bought back to the system after repairing, the internal sequential state information can be obtained from nearby operating modules within one clock cycle. This method is not restricted by the number of redundant modules whereas the traditional state synchronization method can only be applied a system with an odd number of modules.

7.2.5 HDL converter

A Verilog compiler based software for HDL file modification is introduced in this thesis. This software is capable of automatically modifying the user logic and upgrading it into a more reliable version by applying the self-repairing architecture. This method eliminates the human interaction, therefore increases the productivity while avoiding the human error.

Although the HDL converter presented in this thesis is designed for only deploying the predefined self-repairing architecture, this compiler based code modification software has potential in various applications. For example, by customizing the AST modification rules, users can easily insert related logic for increasing the circuit testability.

7.2.6 Self-Repairing Design Framework

This thesis presents a framework for designing and monitoring a self-testable, self-correcting, self-repairing and self-healing digital systems. The self-repairing architecture significantly increases the system reliability without compromising the system availability. In addition, this architecture is automatically deployed by the HDL converter. Furthermore, by ignoring the transient errors, the healing controller can effectively repair the soft permanent errors.

7.3 Conclusion

The objective of this research was to establish a systematic approach for the design of self-testable, self-correcting, self-repairing and self-healing digital systems. Experimental results presented in this thesis demonstrate the improvement in system reliability for a variety of designs under random error injection.

The methods and techniques presented in this thesis offer six distinct contributions discussed in the previous section. Theses contributions provides a fully autonomous approach for the design of self-repairing systems. Furthermore, the HDL converter presented in this thesis, even though was originally designed for generating a self-repairing system, is not limited to this research objective. Customization is allowed for future applications.

7.4 Future Work

This thesis demonstrated a systematic approach for designing self-repairing systems. There are some future works that can be done to further improve the system integration. More applications, especially the safety critical applications, tests and reliability tests are also worth investigating to further examining the performance of the system.

7.4.1 Built-in Healing Controller

The healing controller presented in Chapter 4 relies on an external computer for executing the reconfiguration command. However, this external healing controller can be integrated into the same chip based on the types of FPGAs.

For system on chip FPGA, for example Xilinx Zynq FPGA and Intel Cyclone V FPGA, there is an embedded processor which is capable of running the Linux operating system. This allows the device to execute the partial reconfiguration task the same way as an external desktop does.

For FPGAs without an embedded processor, the same task can be achieved by programming the FPGA to implement a soft processor IP core. This soft processor would not achieve the same performance compared with a dedicated hardware processor, but it is still capable of running an embedded operating system, thus producing the same result as the hard processor.

For FPGAs with very limited resources, the implementation of the soft processor can be expensive or even impossible. In this case, a lightweight partial reconfiguration controller can be applied. This controller reads the partial bitstream in flash and programs the target area in a similar way as the FPGA booting from flash memory.

7.4.2 Safety-Critical Application

To further examine the self-repairing system performance, more safety-critical applications can be implemented. Example applications include, but are not limited to, automotive control systems, nuclear reactor control systems, flight craft control systems and financial transaction processing systems.

7.4.3 Power supply noise testing

The power supply is a critical element in digital circuit. Noise on the power supply can cause malfunction in all digital and analog systems. To further demonstrate the robust-

ness of our system, power supply noise testing is required. In this test, noise with different strength could be injected to the power supply, the FPGA operation results would be recorded and compare with the device under normal power supply.

7.4.4 Elevated temperature testing

The temperature of the device working environment affects the circuit behavior in many ways. Elevated temperature could harm the semiconductor component, leading to accelerated aging and unexpected operating errors. Therefore, examining the circuit function under elevated temperature and comparing the results from normal temperature could help us evaluate the system reliability improvement with the self-repairing architecture.

7.4.5 Radiation Testing

Radiation induced error is the dominating source of modern semiconductor devices. Moreover, applications such as aerospace exploration forces the device to be exposed to high radiation environment. Therefore, a radiation test, where the FPGA is operated under high energy beam, is worth investigating.

Appendices

APPENDIX A

SELF-REPAIRING SYSTEM AND BENCHMARK DESIGN VERILOG CODE

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/13/2019 02:46:19 PM
// Design Name:
// Module Name: adderTree
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module adderTree #(
parameter NUM = 256,
parameter DATA_WIDTH = 16,
parameter OUT_WIDTH = 32,
parameter DEPTH = $clog2(NUM),
parameter COUNT = 3,
parameter SUM_WIDTH = DATA_WIDTH+DEPTH,
parameter EXTRA = SUM_WIDTH - OUT_WIDTH,
parameter OUT_WIDTH_M = OUT_WIDTH - 1
    )
(
    input clk,
    input rst_n,
    input [DATA_WIDTH * NUM - 1 : 0] terms_flat,
    input ld_fin,
    output reg valid,
    output reg signed [OUT_WIDTH - 1 : 0] sum
);

reg signed [SUM_WIDTH - 1 : 0] pipeline [2*NUM - 1 : 0];   // Pipeline array
reg [COUNT:0] count;

genvar i;// Pack flat terms
generate
    for (i = NUM; i < 2 * NUM; i = i + 1) begin
        always @ (posedge clk) begin
            //pipeline[i] <= { { DEPTH{tems_flat} }  ,terms_flat[(i-NUM) * DATA_WIDTH +: DATA_WIDTH]};
            pipeline[i] <= { {DEPTH{terms_flat[(i-NUM+1)*DATA_WIDTH-1]} } ,terms_flat[(i-NUM+1) * DATA_WIDTH -1 -: DATA_WIDTH]};
        end
    end
endgenerate
// Add terms logarithmically
generate
    for (i = 1; i < NUM; i = i + 1) begin
        always @ (posedge clk) begin
            pipeline[i] <= pipeline[i*2] + pipeline[i*2 + 1];
        end
    end
endgenerate

reg dly_0, dly_1, dly_2, dly_3, dly_4, dly_5, dly_6, dly_7, dly_8;

always @(posedge clk or negedge rst_n)
    begin
```

```
        if (!rst_n)
            begin
                dly_0 <= 1'b0;
                dly_1 <= 1'b0;
                dly_2 <= 1'b0;
                dly_3 <= 1'b0;
                dly_4 <= 1'b0;
                dly_5 <= 1'b0;
                dly_6 <= 1'b0;
                dly_7 <= 1'b0;
                dly_8 <= 1'b0;
                valid <= 1'b0;

            end
        else
            begin
                dly_0 <= ld_fin;
                dly_1 <= dly_0;
                dly_2 <= dly_1;
                dly_3 <= dly_2;
                dly_4 <= dly_3;
                dly_5 <= dly_4;
                dly_6 <= dly_5;
                dly_7 <= dly_6;
                dly_8 <= dly_7;
                valid <= dly_8;
            end
    end
generate
if (EXTRA>0) begin // reduce result size to fit output
    always @(posedge clk or negedge rst_n)
        begin
          if (!rst_n) sum <= { SUM_WIDTH{1'b0} };
          else if (valid==1'b1)
            begin
                if (pipeline[l][SUM_WIDTH - 1] == 1'b0)
                    begin
                        if ( pipeline[l][SUM_WIDTH - 1 -: EXTRA] == {EXTRA{1'b0}} ) //no overflow
                            sum <= pipeline[l][OUT_WIDTH-1:0];
                        else // overflow, return maximum positive number
                            sum <= {1'b0, {OUT_WIDTH_M{1'b1}} };
                    end
                else
                    begin
                      if ( pipeline[l][SUM_WIDTH - 1 -: EXTRA] == {EXTRA{1'b1}} ) //no underflow
                          sum <= pipeline[l][OUT_WIDTH-1:0];
                      else // underflow, return minimum negative number
                          sum <= {1'b1, {OUT_WIDTH_M{1'b0}} };
                    end
            end
        end
    end
else begin // no reduction
    always @(posedge clk or negedge rst_n)
        begin
            if (!rst_n) sum <= { SUM_WIDTH{1'b0} };
            else if (valid==1'b1) sum <= pipeline[l];
        end
end
endgenerate

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/09/2017 09:54:51 AM
// Design Name:
// Module Name: ber
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
```

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module ber(clk, ber_en, errors, rst_n, bit1);
    input clk;
    input ber_en;
    input errors;
    input rst_n;
    output reg bit1;

    reg bit0;

    always @(posedge clk or negedge rst_n)
    begin
    if(!rst_n)
        begin
            bit0 <= 1'b0;
            bit1 <= 1'b0;
        end
    else if(errors && ber_en)
        begin
            if (bit1 == 1'b0 && bit0 == 1'b0)
                begin
                    bit0 <= 1'b1;
                    bit1 <= 1'b0;
                end
            else if (bit1 == 1'b0 && bit0 == 1'b1)
                begin
                    bit0 <= 1'b0;
                    bit1 <= 1'b1;
                end
            else if (bit1 == 1'b1 && bit0 == 1'b0)
                begin
                    bit0 <= 1'b1;
                    bit1 <= 1'b1;
                end
            else
                begin
                    bit0 <= bit0;
                    bit1 <= bit1;
                end
        end
    end
endmodule


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2017 04:24:46 PM
// Design Name:
// Module Name: diagnostic
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module diagnostic(
    input [3:0] inputs,
    input clk,
    output reg[1:0] error_code,
    output reg error
);
```

```
always @(posedge clk)
begin
    case(error_code)
    // no error so far
    2'b11:
    begin
        case(inputs[2:0])
        //no error
        3'b000:
            begin
                error_code <= 2'b11;
                error <= 1'b0;
            end
        3'b111:
            begin
                error_code <= 2'b11;
                error <= 1'b0;
            end
        // c is wrong
        3'b001:
            begin
                error_code <= 2'b00;
                error <= 1'b1;
            end
        3'b110:
            begin
                error_code <= 2'b00;
                error <= 1'b1;
            end
        // b is wrong
        3'b010:
            begin
                error_code <= 2'b01;
                error <= 1'b1;
            end
        3'b101:
            begin
                error_code <= 2'b01;
                error <= 1'b1;
            end
        // a is wrong
        3'b100:
            begin
                error_code <= 2'b10;
                error <= 1'b1;
            end
        3'b011:
            begin
                error_code <= 2'b10;
                error <= 1'b1;
            end
        endcase
    end
    // c was a faulty unit
    2'b00:
    begin
        case(inputs[3:1])
        //no error
        3'b000:
            begin
                error_code <= 2'b00;
                error <= 1'b0;
            end
        3'b111:
            begin
                error_code <= 2'b00;
                error <= 1'b0;
            end
        // c is wrong
        3'b001:
            begin
                error_code <= 2'b01;
                error <= 1'b1;
            end
        3'b110:
            begin
                error_code <= 2'b01;
```

```
            error <= 1'b1;
        end
    // b is wrong
    3'b010:
        begin
            error_code <= 2'b10;
            error <= 1'b1;
        end
    3'b101:
        begin
            error_code <= 2'b10;
            error <= 1'b1;
        end
    // a is wrong
    3'b100:
        begin
            error_code <= 2'b11;
            error <= 1'b1;
        end
    3'b011:
        begin
            error_code <= 2'b11;
            error <= 1'b1;
        end
    endcase
end
// b was a faulty module
2'b01:
begin
    case(inputs[3:2]+inputs[0])
    //no error
    3'b000:
        begin
            error_code <= 2'b01;
            error <= 1'b0;
        end
    3'b111:
        begin
            error_code <= 2'b01;
            error <= 1'b0;
        end
    // c is wrong
    3'b001:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    3'b110:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    // b is wrong
    3'b010:
        begin
            error_code <= 2'b10;
            error <= 1'b1;
        end
    3'b101:
        begin
            error_code <= 2'b10;
            error <= 1'b1;
        end
    // a is wrong
    3'b100:
        begin
            error_code <= 2'b11;
            error <= 1'b1;
        end
    3'b011:
        begin
            error_code <= 2'b11;
            error <= 1'b1;
        end
    endcase
end
// a was a faulty module
2'b10:
```

```
begin
    case(inputs[3]+inputs[1:0])
    //no error
    3'b000:
        begin
            error_code <= 2'b10;
            error <= 1'b0;
        end
    3'b111:
        begin
            error_code <= 2'b10;
            error <= 1'b0;
        end
    // c is wrong
    3'b001:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    3'b110:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    // b is wrong
    3'b010:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    3'b101:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    // a is wrong
    3'b100:
        begin
            error_code <= 2'b11;
            error <= 1'b1;
        end
    3'b011:
        begin
            error_code <= 2'b11;
            error <= 1'b1;
        end
    endcase
end
default:
begin
    case(inputs[2:0])
    //no error
    3'b000:
        begin
            error_code <= 2'b11;
            error <= 1'b0;
        end
    3'b111:
        begin
            error_code <= 2'b11;
            error <= 1'b0;
        end
    // c is wrong
    3'b001:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    3'b110:
        begin
            error_code <= 2'b00;
            error <= 1'b1;
        end
    // b is wrong
    3'b010:
        begin
            error_code <= 2'b01;
            error <= 1'b1;
```

```
                end
            3'b101:
                begin
                    error_code <= 2'b01;
                    error <= 1'b1;
                end
            // a is wrong
            3'b100:
                begin
                    error_code <= 2'b10;
                    error <= 1'b1;
                end
            3'b011:
                begin
                    error_code <= 2'b10;
                    error <= 1'b1;
                end
            endcase
        end
        endcase
    end

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/01/2019 03:27:54 PM
// Design Name:
// Module Name: errInject
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// Bit Errors Rate Case: BERC
// ber = 1 / (2 ** (8-BERC) )
// for example BERC==0: ber = 1/256; 7: ber = 1/2


module errInject
#(parameter SEED = 32'h55_5555)
(
input clk,
input rst_n,
input [1:0] inj_mode,
input [3:0] error_rate,
output reg [3:0] output_layer_error,
output reg [319:0] hidden_layer_error
    );
reg [30:0] sh;
always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) sh <= 31'h7fff_ffff;
    else sh <= error_rate - 1'b1;
end



wire [30:0]    lfsr_reg_o;
reg random;
always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) hidden_layer_error <= 320'h0;
    else hidden_layer_error <= {hidden_layer_error[318:0], random};
end

reg [30:0] hl_threshold;
```

```
always @(posedge clk or negedge rst_n)
begin
if (!rst_n)
    begin
        hl_threshold <= 31'h0;
        random <= 1'b0;
    end
else
    begin
        //hl_threshold <= 31'h1 << sh;
//        hl_threshold <= {8'h0,error_rate,19'h0};
hl_threshold <= {4'h0,error_rate,23'h0}; // for simulatoin waveform
        if (lfsr_reg_o < hl_threshold) random <= 1'b1;
        else random <= 1'b0;
    end
end



always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) output_layer_error <= 4'h0;
    else output_layer_error <= {output_layer_error[2:0], random};
end




    lfsr #(.SEED(SEED))
    err_index_gen(
    .clk(clk), // input clock
    .rst_n(rst_n), // synchronous reset
    .lfsr_reg_o(lfsr_reg_o)
    );


endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/18/2019 09:50:23 PM
// Design Name:
// Module Name: hidden_layer_v2
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module hidden_layer_v2 #(
parameter UNIT_NUM = 49,
parameter NODE_NUM = 10,
parameter OUT_WIDTH = 32,
parameter DATA_WIDTH = 8
)(
    input clk,
    input rst_n,
    input new,
    input [UNIT_NUM*DATA_WIDTH-1:0] data,
    input [UNIT_NUM*8-1:0] weights_0,
    input [UNIT_NUM*8-1:0] weights_1,
    input [UNIT_NUM*8-1:0] weights_2,
    input [UNIT_NUM*8-1:0] weights_3,
    input [UNIT_NUM*8-1:0] weights_4,
    input [UNIT_NUM*8-1:0] weights_5,
    input [UNIT_NUM*8-1:0] weights_6,
    input [UNIT_NUM*8-1:0] weights_7,
```

```
    input [UNIT_NUM*8-1:0] weights_8,
    input [UNIT_NUM*8-1:0] weights_9,
    input ld_finish,
    output [NODE_NUM*OUT_WIDTH-1:0] results,
    output valid // need to transfter to pulse before sending to uart
    );


node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N0 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_0),
.ld_fin(ld_finish),
.node_valid(valid),
.node_result(results[OUT_WIDTH-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N1 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_1),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*2-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N2 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_2),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*3-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N3 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_3),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*4-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N4 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_4),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*5-1 -: OUT_WIDTH])
);
```

```
node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N5 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_5),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*6-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N6 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_6),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*7-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N7 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_7),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*8-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N8 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_8),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*9-1 -: OUT_WIDTH])
);

node_top #(
.UNIT_NUM(UNIT_NUM),
.DATA_WIDTH(DATA_WIDTH),
.OUT_WIDTH(OUT_WIDTH))
N9 (
.clk(clk),
.rst_n(rst_n),
.new(new),
.data(data),
.weight(weights_9),
.ld_fin(ld_finish),
.node_result(results[OUT_WIDTH*10-1 -: OUT_WIDTH])
);


endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
```

```
//
// Create Date: 04/01/2019 02:39:01 PM
// Design Name:
// Module Name: lfsr
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module lfsr #(parameter SEED = 31'h55_5555)(
input                 clk, // input clock
input               rst_n, // synchronous reset
output [30:0]    lfsr_reg_o
    );

   reg [30:0]               lfsr_reg_r;
    wire [30:0]              next_lfsr_reg;

always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) lfsr_reg_r <= SEED;
    else lfsr_reg_r <= next_lfsr_reg;//next_lfsr_reg;
end

assign lfsr_reg_o = lfsr_reg_r;


assign next_lfsr_reg[30] = lfsr_reg_r[28];
assign next_lfsr_reg[29] = lfsr_reg_r[27];
assign next_lfsr_reg[28] = lfsr_reg_r[26];
assign next_lfsr_reg[27] = lfsr_reg_r[25];
assign next_lfsr_reg[26] = lfsr_reg_r[24];
assign next_lfsr_reg[25] = lfsr_reg_r[23];
assign next_lfsr_reg[24] = lfsr_reg_r[22];
assign next_lfsr_reg[23] = lfsr_reg_r[21];
assign next_lfsr_reg[22] = lfsr_reg_r[20];
assign next_lfsr_reg[21] = lfsr_reg_r[19];
assign next_lfsr_reg[20] = lfsr_reg_r[18];
assign next_lfsr_reg[19] = lfsr_reg_r[17];
assign next_lfsr_reg[18] = lfsr_reg_r[16];
assign next_lfsr_reg[17] = lfsr_reg_r[15];
assign next_lfsr_reg[16] = lfsr_reg_r[14];
assign next_lfsr_reg[15] = lfsr_reg_r[13];
assign next_lfsr_reg[14] = lfsr_reg_r[12];
assign next_lfsr_reg[13] = lfsr_reg_r[11];
assign next_lfsr_reg[12] = lfsr_reg_r[10];
assign next_lfsr_reg[11] = lfsr_reg_r[9];
assign next_lfsr_reg[10] = lfsr_reg_r[8];
assign next_lfsr_reg[9] = lfsr_reg_r[7];
assign next_lfsr_reg[8] = lfsr_reg_r[6];
assign next_lfsr_reg[7] = lfsr_reg_r[5];
assign next_lfsr_reg[6] = lfsr_reg_r[4];
assign next_lfsr_reg[5] = lfsr_reg_r[3];
assign next_lfsr_reg[4] = lfsr_reg_r[2];
assign next_lfsr_reg[3] = lfsr_reg_r[1];
assign next_lfsr_reg[2] = lfsr_reg_r[0];
assign next_lfsr_reg[1] = ~(lfsr_reg_r[30] ^ lfsr_reg_r[27]);
assign next_lfsr_reg[0] = ~(lfsr_reg_r[29] ^ lfsr_reg_r[26]);


endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/11/2019 04:45:14 PM
// Design Name:
```

```
// Module Name: max
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module max (
            input clk,
            input [3:0] a_index,
            input [3:0] b_index,
            input signed [31:0] a,
            input signed [31:0] b,
            output reg signed [31:0] out,
            output reg [3:0] index);

  always @(posedge clk)
        begin
        if (a>=b) begin
            out <= a;
            index <= a_index;
        end else begin
            out <= b;
            index <= b_index;
        end
  end

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/11/2019 04:45:14 PM
// Design Name:
// Module Name: maxlayer
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module maxlayer(
input clk,
input en,
output reg valid,
input [32*10-1:0] gemm,
output reg [3:0] pred
    );

wire signed [31:0] a0, a1, a2, a3, a4, a5, a6, a7, a8, a9;
wire signed [31:0] m01, m23, m45, m67, m89, mL10, mL11, mL20, mL30;
wire en_m01, en_m23, en_m45, en_m67, en_m89, en_mL10, en_mL11, en_mL20, en_mL30;
wire [3:0] maxid_01, maxid_23, maxid_45, maxid_67, maxid_89,
maxid_L10, maxid_L11,
maxid_L20,
maxid_L30;


assign a9=gemm[32*10-1: 32*9];
assign a8=gemm[32*9-1: 32*8];
assign a7=gemm[32*8-1: 32*7];
```

```
assign a6=gemm[32*7-1: 32*6];
assign a5=gemm[32*6-1: 32*5];
assign a4=gemm[32*5-1: 32*4];
assign a3=gemm[32*4-1: 32*3];
assign a2=gemm[32*3-1: 32*2];
assign a1=gemm[32*2-1: 32*1];
assign a0=gemm[32*1-1: 32*0];


reg valid_0,valid_1,valid_2,valid_3,valid_4,valid_5,valid_6,valid_7;
always @(posedge clk)
begin
    valid_0 <= en;
    valid_1 <= valid_0;
    valid_2 <= valid_1;
    valid_3 <= valid_2;
    valid_4 <= valid_3;
    valid_5 <= valid_4;
    valid_6 <= valid_5;
    valid_7 <= valid_6;
    valid   <= valid_7;
end


always @(posedge clk)
begin
    if (valid) pred <= maxid_L30;
end
reg [3:0] index0  = 4'h0;
reg [3:0] index1  = 4'h1;
reg [3:0] index2  = 4'h2;
reg [3:0] index3  = 4'h3;
reg [3:0] index4  = 4'h4;
reg [3:0] index5  = 4'h5;
reg [3:0] index6  = 4'h6;
reg [3:0] index7  = 4'h7;
reg [3:0] index8  = 4'h8;
reg [3:0] index9  = 4'h9;

//
    max max01(
    .clk(clk),
    .a_index(index0),
    .b_index(index1),
    .a(a0),
    .b(a1),
    .out(m01),
    .index(maxid_01));

    max max23(.clk(clk),
    .a_index(index2),
    .b_index(index3),
    .a(a2),
    .b(a3),
    .out(m23),
    .index(maxid_23));

    max max45(.clk(clk),
    .a_index(index4),
    .b_index(index5),
    .a(a4),
    .b(a5),
    .out(m45),
    .index(maxid_45));

    max max67(.clk(clk),
    .a_index(index6),
    .b_index(index7),
    .a(a6),
    .b(a7),
    .out(m67),
    .index(maxid_67));

    max max89(.clk(clk),
    .a_index(index8),
    .b_index(index9),
    .a(a8),
    .b(a9),
```

```
    .out(m89),
    .index(maxid_89));

    max maxL10(.clk(clk),
    .a_index(maxid_01),
    .b_index(maxid_23),
    .a(m01),
    .b(m23),
    .out(mL10),
    .index(maxid_L10));


    max maxL11(.clk(clk),
    .a_index(maxid_45),
    .b_index(maxid_67),
    .a(m45),
    .b(m67),
    .out(mL11),
    .index(maxid_L11));

    max maxL20(.clk(clk),
    .a_index(maxid_L11),
    .b_index(maxid_89),
    .a(mL11),
    .b(m89),
    .out(mL20),
    .index(maxid_L20));

    max maxL30(.clk(clk),
    .a_index(maxid_L10),
    .b_index(maxid_L20),
    .a(mL10),
    .b(mL20),
    .out(mL30),
    .index(maxid_L30));




endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/11/2019 04:45:14 PM
// Design Name:
// Module Name: node
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module mul #(
parameter DATA_WIDTH = 8,
parameter EXTRA = 1
)(
    input clk,
    input rst_n,
    input signed [DATA_WIDTH-1:0] data,
    input signed [7:0] weight,
    output reg signed [DATA_WIDTH+7:0] product
    );


  always @(posedge clk)
      begin
```

```
        if (!rst_n) product <= 32'h0;
        else product <= data * weight;
      end

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/19/2019 12:43:18 AM
// Design Name:
// Module Name: nn_v2
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
`define INPUT_NUM 784
`define UNIT_NUM 49
`define NODE_NUM 10
`define LAYER_OUT_WIDTH 32
`define DATA_WIDTH 8

module nn_v2(
    input clk,
    input rst_n,
    input en_ld,
    input clear,
    input [1:0] inj_mode,
    input [`DATA_WIDTH-1:0] input_data,
    input [7:0] node_weight_0,
    input [7:0] node_weight_1,
    input [7:0] node_weight_2,
    input [7:0] node_weight_3,
    input [7:0] node_weight_4,
    input [7:0] node_weight_5,
    input [7:0] node_weight_6,
    input [7:0] node_weight_7,
    input [7:0] node_weight_8,
    input [7:0] node_weight_9,
    input [`LAYER_OUT_WIDTH*`NODE_NUM-1:0] hidden_layer_error,
    input [3:0] output_layer_error,
    output reg nn_valid,
    output reg [3:0] pred
    );

wire [`LAYER_OUT_WIDTH*`NODE_NUM-1:0] L1_results;

reg [`UNIT_NUM*`DATA_WIDTH-1:0] temp_data;
reg [`UNIT_NUM*`DATA_WIDTH-1:0] image;
reg [`UNIT_NUM*8-1:0] layer_weights_0;
reg [`UNIT_NUM*8-1:0] layer_weights_1;
reg [`UNIT_NUM*8-1:0] layer_weights_2;
reg [`UNIT_NUM*8-1:0] layer_weights_3;
reg [`UNIT_NUM*8-1:0] layer_weights_4;
reg [`UNIT_NUM*8-1:0] layer_weights_5;
reg [`UNIT_NUM*8-1:0] layer_weights_6;
reg [`UNIT_NUM*8-1:0] layer_weights_7;
reg [`UNIT_NUM*8-1:0] layer_weights_8;
reg [`UNIT_NUM*8-1:0] layer_weights_9;

reg [`UNIT_NUM*8-1:0] temp_weights_0;
reg [`UNIT_NUM*8-1:0] temp_weights_1;
reg [`UNIT_NUM*8-1:0] temp_weights_2;
reg [`UNIT_NUM*8-1:0] temp_weights_3;
reg [`UNIT_NUM*8-1:0] temp_weights_4;
reg [`UNIT_NUM*8-1:0] temp_weights_5;
reg [`UNIT_NUM*8-1:0] temp_weights_6;
reg [`UNIT_NUM*8-1:0] temp_weights_7;
```

```
reg [`UNIT_NUM*8-1:0] temp_weights_8;
reg [`UNIT_NUM*8-1:0] temp_weights_9;
reg [31:0] i;
reg load_data_finished;
reg load_weights_finished;
reg new;
wire max_en;
reg max_en_dly, max_en_dly2;

always @(posedge clk)
begin
    max_en_dly <= max_en;
    max_en_dly2 <= max_en_dly;
end


hidden_layer_v2 #(
    .UNIT_NUM(`UNIT_NUM),
    .NODE_NUM(`NODE_NUM),
    .DATA_WIDTH(`DATA_WIDTH),
    .OUT_WIDTH(`LAYER_OUT_WIDTH))
    layer1 (
    .clk(clk),
    .rst_n(rst_n),
    .new(clear),
    .data(image),
    .weights_0(layer_weights_0),
    .weights_1(layer_weights_1),
    .weights_2(layer_weights_2),
    .weights_3(layer_weights_3),
    .weights_4(layer_weights_4),
    .weights_5(layer_weights_5),
    .weights_6(layer_weights_6),
    .weights_7(layer_weights_7),
    .weights_8(layer_weights_8),
    .weights_9(layer_weights_9),
    .ld_finish(load_data_finished),
    .valid(max_en),
    .results(L1_results)
);
wire maxlayer_valid;
reg [`LAYER_OUT_WIDTH*`NODE_NUM-1:0] L1_results_with_err;

/*
    errInject #(.BERC(6))
    error_inject (
    .clk(clk),
    .en(err_en),
    .rst_n(rst_n),
    .din(L1_results),
    .dout(L1_results_with_err)
    );
*/


// inj_mode 01: hidden layer error only; 10 outputlayer only;, 11 all;
wire [3:0] output_layer_result;
always @(posedge clk)
begin
    if (inj_mode[0] == 1'b1) L1_results_with_err <= L1_results ^ hidden_layer_error;
    else L1_results_with_err <= L1_results;
end

always @(posedge clk)
begin
    if (inj_mode[1] == 1'b1) pred <= output_layer_result ^ output_layer_error;
    else pred <= output_layer_result;
end


maxlayer max(
    .clk(clk),
    .en(max_en_dly2),
    .gemm(L1_results_with_err),
    .pred(output_layer_result),
    .valid(maxlayer_valid)
);
always @(posedge clk)
```

```
begin
    nn_valid <= maxlayer_valid;
end


reg en_ld_dly;

always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) en_ld_dly <= 1'b0;
    else en_ld_dly <= en_ld;
end


always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) new <= 1'b0;
    else if (en_ld == 1'b0 && en_ld_dly == 1'b1) new <= en_ld;
    else new <= 1'b0;
end




always @(posedge clk)
    begin
        if (en_ld | en_ld_dly)// && ~load_weights_finished)
        begin
            temp_weights_0 <= {node_weight_0, temp_weights_0[`UNIT_NUM*8-1 : 8] };
            temp_weights_1 <= {node_weight_1, temp_weights_1[`UNIT_NUM*8-1 : 8] };
            temp_weights_2 <= {node_weight_2, temp_weights_2[`UNIT_NUM*8-1 : 8] };
            temp_weights_3 <= {node_weight_3, temp_weights_3[`UNIT_NUM*8-1 : 8] };
            temp_weights_4 <= {node_weight_4, temp_weights_4[`UNIT_NUM*8-1 : 8] };
            temp_weights_5 <= {node_weight_5, temp_weights_5[`UNIT_NUM*8-1 : 8] };
            temp_weights_6 <= {node_weight_6, temp_weights_6[`UNIT_NUM*8-1 : 8] };
            temp_weights_7 <= {node_weight_7, temp_weights_7[`UNIT_NUM*8-1 : 8] };
            temp_weights_8 <= {node_weight_8, temp_weights_8[`UNIT_NUM*8-1 : 8] };
            temp_weights_9 <= {node_weight_9, temp_weights_9[`UNIT_NUM*8-1 : 8] };
        end
    end



always @(posedge clk)
begin
    if (en_ld  | en_ld_dly)//&& ~load_data_finished)
    begin
        temp_data <= {{1'b0,input_data[`DATA_WIDTH-1:1]}, temp_data[`UNIT_NUM*`DATA_WIDTH-1 : `DATA_WIDTH]};
    end
end

always @(posedge clk)
begin
    if ( (i % `UNIT_NUM == 0) && (i > 0) )
    begin
        image <= temp_data;
        load_data_finished <= 1'b1;
    end
    else
    begin
        image <= image;
        load_data_finished <= 1'b0;
    end
end



always @(posedge clk)
begin
   if (i % `UNIT_NUM == 0 && i > 0)
    begin
        layer_weights_0 <= temp_weights_0;
        layer_weights_1 <= temp_weights_1;
        layer_weights_2 <= temp_weights_2;
        layer_weights_3 <= temp_weights_3;
        layer_weights_4 <= temp_weights_4;
        layer_weights_5 <= temp_weights_5;
        layer_weights_6 <= temp_weights_6;
```

```
        layer_weights_7 <= temp_weights_7;
        layer_weights_8 <= temp_weights_8;
        layer_weights_9 <= temp_weights_9;
        load_weights_finished <= 1'b1;
    end
    else
    begin
        layer_weights_0 <= layer_weights_0;
        layer_weights_1 <= layer_weights_1;
        layer_weights_2 <= layer_weights_2;
        layer_weights_3 <= layer_weights_3;
        layer_weights_4 <= layer_weights_4;
        layer_weights_5 <= layer_weights_5;
        layer_weights_6 <= layer_weights_6;
        layer_weights_7 <= layer_weights_7;
        layer_weights_8 <= layer_weights_8;
        layer_weights_9 <= layer_weights_9;
        load_weights_finished <= 1'b0;
    end
end
/*
always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) i <= 32'h0;
    else if ( en_ld ) i <= 32'hffff_ffff;
    else i <= i + 1'b1;
end


reg loading;
always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) loading <= 1'b0;
    else if ( en_ld ) loading <= 1'b1;
    else if (i == 32'd783) loading <= 1'b0;
end
*/

always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) i <= 32'hffff_ffff;
    else if (en_ld | en_ld_dly) i <= i + 1'b1;
    else i <= 32'hffff_ffff;
end


endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/18/2019 10:51:22 PM
// Design Name:
// Module Name: node_top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module node_top #(
parameter UNIT_NUM = 49,
parameter OUT_WIDTH = 32,
parameter DATA_WIDTH = 8
)(
    input clk,
    input rst_n,
    input new, // pulse to reset unit_valid counter
```

```
    input [UNIT_NUM*DATA_WIDTH-1:0] data,
    input [UNIT_NUM*8-1:0] weight,
    input ld_fin,
    output reg node_valid,
    output reg signed [OUT_WIDTH-1:0] node_result
    );
reg signed [OUT_WIDTH-1:0] result;
wire [OUT_WIDTH-1:0] partial_sum;
wire unit_valid;
reg [4:0] count; // 784 / 49 = 16
reg unit_rst_n;

always @(posedge clk)
begin
    unit_rst_n <= rst_n;
end

unit #(
    .NUM(UNIT_NUM),
    .DATA_WIDTH(DATA_WIDTH),
    .OUT_WIDTH(OUT_WIDTH))
    N (
    .clk(clk),
    .rst_n(rst_n),
    .data(data),
    .weight(weight),
    .ld_fin(ld_fin),
    .valid(unit_valid),
    .result(partial_sum)
    );




reg unit_valid_dly0, unit_valid_dly1, unit_valid_dly2, unit_valid_dly3, unit_valid_dly4;
always @(posedge clk)
    begin
        unit_valid_dly0 <= unit_valid;
        unit_valid_dly1 <= unit_valid_dly0;
        unit_valid_dly2 <= unit_valid_dly1;
        unit_valid_dly3 <= unit_valid_dly2;
        unit_valid_dly4 <= unit_valid_dly3;
    end


always @(posedge clk or negedge rst_n)
    begin
        if (!rst_n) count <= 'b0;
        else if (unit_valid_dly4) count <= count + 1'b1;
        else if (node_valid) count <= 'b0;
    end

always @(posedge clk or negedge rst_n)
    begin
        if (!rst_n) result <= { OUT_WIDTH{1'b0} };
        else if (new) result <= { OUT_WIDTH{1'b0} };
        else if (unit_valid_dly4) result <= result + partial_sum;
    end

always @(posedge clk or negedge rst_n)
        begin
            if (!rst_n) node_valid <= 1'b0;
            else if (count == 5'h10) node_valid <= 1'b1;
            else node_valid <= 1'b0;
        end



always @(posedge clk or negedge rst_n)
        begin
            if (!rst_n) node_result <= { OUT_WIDTH{1'b0} };
            else if (node_valid) node_result <= result;
        end


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/18/2019 10:15:50 PM
// Design Name:
// Module Name: node_v2
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module unit #(
parameter NUM = 49,
parameter DATA_WIDTH = 8,
parameter OUT_WIDTH = 32,
parameter EXTRA = $clog2(NUM),
//parameter DEPTH = $clog2(EXTRA),
//parameter SUM_WIDTH = DATA_WIDTH+8+EXTRA
//parameter N1_DATA_WIDTH =8,
parameter ADDER_WIDTH = DATA_WIDTH + 8 //DATA_WIDTH + WEIGHT_WIDTH
)(
    input clk,
    input rst_n,
    input [NUM*DATA_WIDTH-1:0] data,
    input [NUM*8-1:0] weight,
    input ld_fin,
    output valid,
    output signed [OUT_WIDTH-1:0] result
    );
wire [ADDER_WIDTH-1:0] product [0:NUM];
wire [ADDER_WIDTH * NUM - 1 : 0] product_flat;
reg mul_rst_n [0:NUM];
reg adder_rst_n;

always @(posedge clk)
begin
    adder_rst_n <= rst_n;
end

//wire [SUM_WIDTH:0] temp [EXTRA-1:0];
//reg signed [SUM_WIDTH:0] sum;

genvar gi;
generate

for (gi=0; gi<NUM; gi=gi+1)
    begin: multi
    always @(posedge clk)
    begin
        mul_rst_n[gi] <= rst_n;
    end
        //wire [DATA_WIDTH+7:0] product;
        mul #(
            .DATA_WIDTH(DATA_WIDTH))
            inst (
            .clk(clk),
            .rst_n(rst_n),
            .data(data[ DATA_WIDTH*(gi+1) - 1 -: DATA_WIDTH ]),
            .weight(weight[8*(gi+1) - 1 -: 8]),
            .product(product[gi])
        );
    assign product_flat[ADDER_WIDTH*(gi+1) - 1 -: ADDER_WIDTH] = product[gi];
    end
endgenerate
adderTree #(
    .NUM(NUM),
    .DATA_WIDTH(ADDER_WIDTH),
```

```
    .OUT_WIDTH(OUT_WIDTH)
    )
    sum (
    .clk(clk),
    .rst_n(rst_n),
    .terms_flat(product_flat),
    .ld_fin(ld_fin),
    .valid(valid),
    .sum(result)
    );


endmodule


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/08/2017 09:58:10 PM
// Design Name:
// Module Name: qmr_enh
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module qmr_enh(
    input clk,
    input ber_en,
    input [3:0] inputs,
    input ber_rstn,
    output out,
    output master_err,
    output [1:0] err_code
);

    wire error;

    voting vote(
    .clk(clk),
    .inputs(inputs),
    .input_err_code(err_code),
    .v(out)
    );

    diagnostic diagno(
    .clk(clk),
    .inputs(inputs),
    .error_code(err_code),
    .error(error)
    );

    ber error_counter(
    .clk(clk),
    .ber_en(ber_en),
    .errors(error),
    .rst_n(ber_rstn),
    .bit1(master_err)
    );


endmodule


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
```

```
//
// Create Date: 04/04/2019 12:58:43 AM
// Design Name:
// Module Name: top_qmr
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////



`define c_CLKS_PER_BIT 1736

module top_qmr(
input SYSCLK_N,
input SYSCLK_P,
input USB_TX,
input GPIO_DIP_SW3,
input GPIO_DIP_SW2,
input GPIO_DIP_SW1,
input GPIO_DIP_SW0,
input GPIO_SW_N,
input GPIO_SW_S,
//input GPIO_SW_E,
output GPIO_LED_7_LS,
output GPIO_LED_6_LS,
output GPIO_LED_5_LS,
output GPIO_LED_4_LS,
output GPIO_LED_3_LS,
output GPIO_LED_2_LS,
output GPIO_LED_1_LS,
output GPIO_LED_0_LS,
output USB_RX
);
wire uart_clk;
// divide uart_clk by 16 to get cell_clk
wire cell_clk;
reg reseting;
wire uart_tx_done;
wire sys_rst, sys_rst_n;
wire rst_n_wire;

assign sys_rst = GPIO_SW_N;
assign sys_rst_n = ~ sys_rst;
wire rst_n;
assign rst_n =  ~ sys_rst;
/*
reg rst_n;
assign rst_n_wire = ~ (sys_rst | reseting);
always @(posedge uart_clk)
begin
    rst_n <= rst_n_wire;
end
*/
IBUFGDS #(
    .DIFF_TERM("FALSE"), // Differential Termination
    .IBUF_LOW_PWR("TRUE"), // Low power="TRUE", Highest performance="FALSE"
    .IOSTANDARD("DEFAULT") // Specify the input I/O standard
    ) IBUFGDS_inst (
    .O(uart_clk), // Clock buffer output
    .I(SYSCLK_P), // Diff_p clock buffer input (connect directly to top-level port)
    .IB(SYSCLK_N) // Diff_n clock buffer input (connect directly to top-level port)
    );
  clk_wiz_1 clk_gen
     (
      // Clock out ports
      .clk_out1(cell_clk),     // output clk_out1
      // Status and control signals
      .resetn(sys_rst_n), // input resetn
      .locked(GPIO_LED_3_LS),       // output locked
```

```
     // Clock in ports
      .clk_in1(uart_clk));

wire rx_fifo_empty;
wire tx_fifo_empty;
//////// debug /////////////////
assign GPIO_LED_6_LS = rx_fifo_empty;
assign GPIO_LED_4_LS = tx_fifo_empty;


////////// debug end //////////////

wire [7:0] rx_byte;
wire [7:0] tx_byte;
wire [7:0] tx_data, rx_data;


//reg valid_tx;
wire valid_rx;


reg [7:0] temp_weight_0;
reg [7:0] temp_weight_1;
reg [7:0] temp_weight_2;
reg [7:0] temp_weight_3;
reg [7:0] temp_weight_4;
reg [7:0] temp_weight_5;
reg [7:0] temp_weight_6;
reg [7:0] temp_weight_7;
reg [7:0] temp_weight_8;
reg [7:0] temp_weight_9;
wire nn_valid;
wire [3:0] pred;
reg [7:0] pred_byte;

reg signed [7:0] node_mem_weights_0 [0:783];
reg signed [7:0] node_mem_weights_1 [0:783];
reg signed [7:0] node_mem_weights_2 [0:783];
reg signed [7:0] node_mem_weights_3 [0:783];
reg signed [7:0] node_mem_weights_4 [0:783];
reg signed [7:0] node_mem_weights_5 [0:783];
reg signed [7:0] node_mem_weights_6 [0:783];
reg signed [7:0] node_mem_weights_7 [0:783];
reg signed [7:0] node_mem_weights_8 [0:783];
reg signed [7:0] node_mem_weights_9 [0:783];

initial begin
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_0.mem", node_mem_weights_0, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_1.mem", node_mem_weights_1, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_2.mem", node_mem_weights_2, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_3.mem", node_mem_weights_3, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_4.mem", node_mem_weights_4, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_5.mem", node_mem_weights_5, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_6.mem", node_mem_weights_6, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_7.mem", node_mem_weights_7, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_8.mem", node_mem_weights_8, 0, 783);
    $readmemb("C:/Users/jyang356/Documents/phd_research/itc2019/mnist_nn/node_9.mem", node_mem_weights_9, 0, 783);
end


wire prog_full;
reg [31:0] i;
reg prog_full_ext = 1'b0;

reg [7:0] delay_count;
always @(posedge uart_clk)
begin
    if (prog_full)  delay_count <= 8'b0;
    else delay_count <= delay_count + 1'b1;
end

always @(posedge uart_clk)
begin
    if (prog_full)  prog_full_ext <= 1'b1;
    else if (delay_count == 8'hff) prog_full_ext <= 1'b0;
end

reg prog_full_syn , prog_full_syn2;
```

```
always @(posedge cell_clk)
begin
prog_full_syn <= prog_full_ext;
prog_full_syn2 <= prog_full_syn;
end

reg start_count, pulse, pulse_dly;
always @(posedge cell_clk or negedge rst_n)
begin
    if (!rst_n)
        begin
            pulse <= 1'b0;
            pulse_dly <= 1'b0;
            start_count <= 1'b0;
        end
    else
        begin
            if (prog_full_syn2 == 1'b1) pulse <= 1'b1;
            else                        pulse <= 1'b0;
            pulse_dly <= pulse;
            if (pulse == 1'b1 && pulse_dly == 1'b0) start_count <= 1'b1;
            else start_count <= 1'b0;
        end
end


always @(posedge cell_clk or negedge rst_n)
begin
    if (!rst_n) i <= 32'h0;
    else if ( start_count ) i <= 32'h0;
    else i <= i + 1'b1;
end


reg loading=1'b0;
always @(posedge cell_clk)
begin
    if ( start_count ) loading <= 1'b1;
    else if (i == 32'd783) loading <= 1'b0;
    //else loading <= 1'b0;
end


reg [9:0] j;
always @ (posedge uart_clk or negedge sys_rst_n)
begin
    if (!sys_rst_n) j <= 10'b0;
    else if ( uart_tx_done ) j <= 10'b0;
    else if ( reseting ) j <= j + 1'b1;
    else j <= 10'b0;
end

always @ (posedge uart_clk or negedge sys_rst_n)
begin
    if (!sys_rst_n) reseting <= 1'b1;
    else if ( uart_tx_done ) reseting <= 1'b1;
    else if ( j == 10'h3_ff ) reseting <= 1'b0;
end



always @(posedge cell_clk)
begin
    if (loading)
    begin
        temp_weight_0 <= node_mem_weights_0[i];
        temp_weight_1 <= node_mem_weights_1[i];
        temp_weight_2 <= node_mem_weights_2[i];
        temp_weight_3 <= node_mem_weights_3[i];
        temp_weight_4 <= node_mem_weights_4[i];
        temp_weight_5 <= node_mem_weights_5[i];
        temp_weight_6 <= node_mem_weights_6[i];
        temp_weight_7 <= node_mem_weights_7[i];
        temp_weight_8 <= node_mem_weights_8[i];
        temp_weight_9 <= node_mem_weights_9[i];
    end
end
```

```
always @(posedge cell_clk)
begin
    pred_byte <= {4'h0,pred};
end

reg loading_dly;
always @(posedge cell_clk  or negedge rst_n)
begin
    if (!rst_n) loading_dly <= 1'b0;
    else loading_dly <= loading;
end

reg nn_valid_flag, nn_valid_flag_dly, nn_valid_flag_dly2,valid_tx;
wire silence;
always @(posedge cell_clk)
begin
    nn_valid_flag <= nn_valid;
    nn_valid_flag_dly <= nn_valid_flag;
    nn_valid_flag_dly2 <= nn_valid_flag_dly;
end

reg fifo_tx_wr_en;
always @(posedge cell_clk)
begin
    if (nn_valid_flag_dly == 1'b1 && nn_valid_flag_dly2 == 1'b0) fifo_tx_wr_en <= 1'b1;
    else fifo_tx_wr_en <= 1'b0;
end

wire [3:0] error_rate;
wire [319:0] hidden_layer_error0, hidden_layer_error1, hidden_layer_error2, hidden_layer_error3;
wire [3:0] output_layer_error0,output_layer_error1, output_layer_error2, output_layer_error3;
//////////extend and sync signal from uart to cell clk domain /////////////////
reg [7:0] repeat_delay_count;
reg uart_tx_done_ext;
always @(posedge uart_clk)
begin
    if (uart_tx_done)  repeat_delay_count <= 8'b0;
    else repeat_delay_count <= repeat_delay_count + 1'b1;
end

always @(posedge uart_clk)
begin
    if (uart_tx_done)  uart_tx_done_ext <= 1'b1;
    else if (repeat_delay_count == 8'hff) uart_tx_done_ext <= 1'b0;
end

reg uart_tx_done_syn , uart_tx_done_syn2;
always @(posedge cell_clk)
begin
uart_tx_done_syn <= uart_tx_done_ext;
uart_tx_done_syn2 <= uart_tx_done_syn;
end
////////////////////////////////////
reg inj_sw;
reg [1:0] inj_mode, rebuf;
//assign GPIO_LED_1_LS = rebuf[1];
//assign GPIO_LED_0_LS = rebuf[0];
assign GPIO_LED_1_LS = inj_mode[1];
assign GPIO_LED_0_LS = inj_mode[0];
//always @(posedge cell_clk)
//begin
//   rebuf <= inj_mode;
//end
reg sw_s_dly;
always @(posedge cell_clk)
begin
    sw_s_dly <= GPIO_SW_S;
    inj_sw <= sw_s_dly;
end

reg inj_sw_pulse, inj_sw_dly;
always @(posedge cell_clk)
begin
    inj_sw_dly <= inj_sw;
end

always @(posedge cell_clk)
```

```
begin
    if (inj_sw == 1'b1 && inj_sw_dly == 1'b0) inj_sw_pulse <= 1'b1;
    else inj_sw_pulse <= 1'b0;
end

always @(posedge cell_clk or negedge rst_n)
begin
    if (!rst_n) inj_mode <= 2'b0;
    else if (inj_sw_pulse == 1'b1) inj_mode <= inj_mode + 1'b1;
end



errInject #(
.SEED(23'h1255)
)
    error_inject0 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .inj_mode(inj_mode),
    .error_rate(error_rate),
    .hidden_layer_error(hidden_layer_error0),
    .output_layer_error(output_layer_error0)
);

errInject #(
.SEED(23'h3478)
)
    error_inject1 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .inj_mode(inj_mode),
    .error_rate(error_rate),
    .hidden_layer_error(hidden_layer_error1),
    .output_layer_error(output_layer_error1)
);

errInject #(
.SEED(23'h5690)
)
    error_inject2 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .inj_mode(inj_mode),
    .error_rate(error_rate),
    .hidden_layer_error(hidden_layer_error2),
    .output_layer_error(output_layer_error2)
);

errInject #(
.SEED(23'h78ab)
)
    error_inject3 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .inj_mode(inj_mode),
    .error_rate(error_rate),
    .hidden_layer_error(hidden_layer_error3),
    .output_layer_error(output_layer_error3)
);

assign error_rate = {GPIO_DIP_SW3, GPIO_DIP_SW2, GPIO_DIP_SW1, GPIO_DIP_SW0};


wire [3:0] pred0, pred1,pred2,pred3;
wire [3:0] in0_qmr, in1_qmr, in2_qmr, in3_qmr;
assign in0_qmr = {pred0[0], pred1[0], pred2[0], pred3[0]};
assign in1_qmr = {pred0[1], pred1[1], pred2[1], pred3[1]};
assign in2_qmr = {pred0[2], pred1[2], pred2[2], pred3[2]};
assign in3_qmr = {pred0[3], pred1[3], pred2[3], pred3[3]};

    (* keep_hierarchy = "yes" *) qmr_enh enhance0(
      .clk(cell_clk),
      .ber_en(nn_valid_flag_dly),
      .inputs(in0_qmr),
      .ber_rstn(rst_n),
      .out(pred[0])
      //.err_code(err_code),
```

```
      //.master_err(master_err)
    );
    (* keep_hierarchy = "yes" *) qmr_enh enhance1(
      .clk(cell_clk),
      .ber_en(nn_valid_flag_dly),
      .inputs(in1_qmr),
      .ber_rstn(rst_n),

      .out(pred[1])
      //.err_code(err_code),
      //.master_err(master_err)
    );
    (* keep_hierarchy = "yes" *) qmr_enh enhance2(
      .clk(cell_clk),
      .ber_en(nn_valid_flag_dly),
      .inputs(in2_qmr),
      .ber_rstn(rst_n),
      .out(pred[2])
      //.err_code(err_code),
      //.master_err(master_err)
    );
    (* keep_hierarchy = "yes" *) qmr_enh enhance3(
      .clk(cell_clk),
      .ber_en(nn_valid_flag_dly),
      .inputs(in3_qmr),
      .ber_rstn(rst_n),
      .out(pred[3])
      //.err_code(err_code),
      //.master_err(master_err)
    );

(* keep_hierarchy = "yes" *) nn_v2 dut0 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .input_data(rx_data),
    .clear(uart_tx_done_syn2),
    .node_weight_0(temp_weight_0),
    .node_weight_1(temp_weight_1),
    .node_weight_2(temp_weight_2),
    .node_weight_3(temp_weight_3),
    .node_weight_4(temp_weight_4),
    .node_weight_5(temp_weight_5),
    .node_weight_6(temp_weight_6),
    .node_weight_7(temp_weight_7),
    .node_weight_8(temp_weight_8),
    .node_weight_9(temp_weight_9),
    .inj_mode(inj_mode),
    .hidden_layer_error(hidden_layer_error0),
    .output_layer_error(output_layer_error0),
//    .hidden_layer_error(320'h0),
//.output_layer_error(4'h0),
    .en_ld(loading),////////////////////////////////////////////////////////////////////
    .nn_valid(nn_valid),
    .pred(pred0)
);

(* keep_hierarchy = "yes" *) nn_v2 dut1 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .input_data(rx_data),
    .clear(uart_tx_done_syn2),
    .node_weight_0(temp_weight_0),
    .node_weight_1(temp_weight_1),
    .node_weight_2(temp_weight_2),
    .node_weight_3(temp_weight_3),
    .node_weight_4(temp_weight_4),
    .node_weight_5(temp_weight_5),
    .node_weight_6(temp_weight_6),
    .node_weight_7(temp_weight_7),
    .node_weight_8(temp_weight_8),
    .node_weight_9(temp_weight_9),
    .inj_mode(inj_mode),
    .hidden_layer_error(hidden_layer_error1),
    .output_layer_error(output_layer_error1),
//    .hidden_layer_error(320'h0),
//.output_layer_error(4'h0),
    .en_ld(loading),////////////////////////////////////////////////////////////////////
    //.nn_valid(nn_valid),
```

```
    .pred(pred1)
);
(* keep_hierarchy = "yes" *) nn_v2 dut2 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .input_data(rx_data),
    .clear(uart_tx_done_syn2),
    .node_weight_0(temp_weight_0),
    .node_weight_1(temp_weight_1),
    .node_weight_2(temp_weight_2),
    .node_weight_3(temp_weight_3),
    .node_weight_4(temp_weight_4),
    .node_weight_5(temp_weight_5),
    .node_weight_6(temp_weight_6),
    .node_weight_7(temp_weight_7),
    .node_weight_8(temp_weight_8),
    .node_weight_9(temp_weight_9),
    .inj_mode(inj_mode),
    .hidden_layer_error(hidden_layer_error2),
    .output_layer_error(output_layer_error2),
//    .hidden_layer_error(320'h0),
//.output_layer_error(4'h0),
    .en_ld(loading),////////////////////////////////////////////////////////////
    //.nn_valid(nn_valid),
    .pred(pred2)
);
(* keep_hierarchy = "yes" *) nn_v2 dut3 (
    .clk(cell_clk),
    .rst_n(rst_n),
    .input_data(rx_data),
    .clear(uart_tx_done_syn2),
    .node_weight_0(temp_weight_0),
    .node_weight_1(temp_weight_1),
    .node_weight_2(temp_weight_2),
    .node_weight_3(temp_weight_3),
    .node_weight_4(temp_weight_4),
    .node_weight_5(temp_weight_5),
    .node_weight_6(temp_weight_6),
    .node_weight_7(temp_weight_7),
    .node_weight_8(temp_weight_8),
    .node_weight_9(temp_weight_9),
    .inj_mode(inj_mode),
    .hidden_layer_error(hidden_layer_error3),
    .output_layer_error(output_layer_error3),
//    .hidden_layer_error(320'h0),
//.output_layer_error(4'h0),
    .en_ld(loading),////////////////////////////////////////////////////////////
    //.nn_valid(nn_valid),
    .pred(pred3)
);
fifo_generator_0 fifo_rx (
  .wr_clk(uart_clk),  // input wire wr_clk
  .rd_clk(cell_clk),  // input wire rd_clk
  .din(rx_byte),        // input wire [7 : 0] din
  .wr_en(valid_rx),    // input wire wr_en
  .rd_en(loading),    // input wire rd_en
  .dout(rx_data),      // output wire [7 : 0] dout
  .full(GPIO_LED_7_LS),// cts?rts?      // output wire full
  .prog_full(prog_full),
  .empty(rx_fifo_empty)    // output wire 15
);

wire fifo_tx_wr_ack, fifo_tx_rd_valid;
reg fifo_tx_wr_ack_dly;
always @(posedge cell_clk)
begin
    fifo_tx_wr_ack_dly <= fifo_tx_wr_ack;
end

reg fifo_tx_wr_done;
always @(posedge cell_clk or negedge rst_n)
begin
    if (!rst_n) fifo_tx_wr_done <= 1'b0;
    else if (fifo_tx_wr_ack==1'b0 && fifo_tx_wr_ack_dly == 1'b1) fifo_tx_wr_done <= 1'b1;
    else fifo_tx_wr_done <= 1'b0;
end

reg tx_fifo_rd_en_in, tx_fifo_rd_en_dly, tx_fifo_rd_en_dly2, tx_fifo_rd_en_dly3, tx_fifo_rd_en;
```

```
always @(posedge uart_clk or negedge rst_n)
begin
    if (!rst_n)
        begin
            tx_fifo_rd_en_in <= 1'b0;
            tx_fifo_rd_en_dly <= 1'b0;
            tx_fifo_rd_en_dly2 <= 1'b0;
            tx_fifo_rd_en_dly3 <= 1'b0;
            tx_fifo_rd_en <= 1'b0;
        end
    else
        begin
            tx_fifo_rd_en_in <= fifo_tx_wr_done;
            tx_fifo_rd_en_dly <= tx_fifo_rd_en_in;
            tx_fifo_rd_en_dly2 <= tx_fifo_rd_en_dly;
            tx_fifo_rd_en_dly3 <= tx_fifo_rd_en_dly2;
            if (tx_fifo_rd_en_dly3 == 1'b1 && tx_fifo_rd_en_dly2 == 1'b0) tx_fifo_rd_en <= 1'b1;
            else tx_fifo_rd_en <= 1'b0;
        end
end


fifo_generator_1 fifo_tx (
  .wr_clk(cell_clk),  // input wire wr_clk
  .rd_clk(uart_clk),  // input wire rd_clk
  .din(pred_byte),        // input wire [7 : 0] din
  .wr_en(fifo_tx_wr_en),    // input wire wr_en
  .rd_en(tx_fifo_rd_en),    // input wire rd_en
  .dout(tx_data),      // output wire [7 : 0] dout
  .full(GPIO_LED_5_LS),      // output wire full
  .wr_ack(fifo_tx_wr_ack),                // output wire wr_ack
  .valid(fifo_tx_rd_valid),                   // output wire valid
  .empty(tx_fifo_empty)    // output wire empty
);




uart_rx #(.CLKS_PER_BIT(`c_CLKS_PER_BIT))
UART_RX_INST
(.i_Clock(uart_clk),
.i_Rx_Serial(USB_TX),
.o_Rx_DV(valid_rx),
.o_Rx_Byte(rx_byte)
);

uart_tx #(.CLKS_PER_BIT(`c_CLKS_PER_BIT))
UART_TX_INST
(.i_Clock(uart_clk),
.i_Tx_DV(fifo_tx_rd_valid),
.i_Tx_Byte(tx_data),
.o_Tx_Active(),
.o_Tx_Serial(USB_RX),
.o_Tx_Done(uart_tx_done)
);



// dubug module ==============================================

/*
load_debug your_instance_name (
        .clk(cell_clk), // input wire clk
        .probe0(loading), // input wire [0:0]  probe0
        .probe1(pulse), // input wire [0:0]  probe1
        .probe2( pred_byte), // input wire [0:0]  probe2
        .probe3(temp_weight_2), // input wire [0:0]  probe3
        .probe4(i) // input wire [31:0]  probe4
);

nn_debug NN_INS (
        .clk(cell_clk), // input wire clk
        .probe0(temp_weight_0), // input wire [7:0]  probe0
        .probe1(temp_weight_1), // input wire [7:0]  probe1
        .probe2(temp_weight_2), // input wire [7:0]  probe2
        .probe3(temp_weight_3), // input wire [7:0]  probe3
        .probe4(temp_weight_4), // input wire [7:0]  probe4
        .probe5(temp_weight_5), // input wire [7:0]  probe5
```

```
        .probe6(temp_weight_6), // input wire [7:0]  probe6
        .probe7(temp_weight_7), // input wire [7:0]  probe7
        .probe8(temp_weight_8), // input wire [7:0]  probe8
        .probe9(temp_weight_9), // input wire [7:0]  probe9
        .probe10(rst_n), // input wire [0:0]  probe10
        .probe11(rx_data), // input wire [7:0]  probe11
        .probe12(loading), // input wire [0:0]  probe12
        .probe13(nn_valid), // input wire [0:0]  probe13
        .probe14(pred) // input wire [3:0]  probe14
);
*/



endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/17/2019 03:26:40 PM
// Design Name:
// Module Name: uart_rx
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module uart_rx
#(parameter CLKS_PER_BIT = 1736,
parameter WIDTH = $clog2(CLKS_PER_BIT),
parameter s_IDLE         = 3'b000,
parameter s_RX_START_BIT = 3'b001,
parameter s_RX_DATA_BITS = 3'b010,
parameter s_RX_STOP_BIT  = 3'b011,
parameter s_CLEANUP      = 3'b100)
  (
   input        i_Clock,
   input        i_Rx_Serial,
   output       o_Rx_DV,
   output [7:0] o_Rx_Byte
   );

  //parameter CLKS_PER_BIT = 87 ;



  reg           r_Rx_Data_R = 1'b1;
  reg           r_Rx_Data   = 1'b1;

  reg [WIDTH:0]     r_Clock_Count = 0;
  reg [2:0]     r_Bit_Index   = 0; //8 bits total
  reg [7:0]     r_Rx_Byte     = 0;
  reg           r_Rx_DV       = 0;
  reg [2:0]     r_SM_Main     = 0;

  // Purpose: Double-register the incoming data.
  // This allows it to be used in the UART RX Clock Domain.
  // (It removes problems caused by metastability)
  always @(posedge i_Clock)
    begin
      r_Rx_Data_R <= i_Rx_Serial;
      r_Rx_Data   <= r_Rx_Data_R;
    end


  // Purpose: Control RX state machine
  always @(posedge i_Clock)
```

```
begin

  case (r_SM_Main)
    s_IDLE :
      begin
        r_Rx_DV       <= 1'b0;
        r_Clock_Count <= 0;
        r_Bit_Index   <= 0;

        if (r_Rx_Data == 1'b0)          // Start bit detected
          r_SM_Main <= s_RX_START_BIT;
        else
          r_SM_Main <= s_IDLE;
      end

    // Check middle of start bit to make sure it's still low
    s_RX_START_BIT :
      begin
        if (r_Clock_Count == (CLKS_PER_BIT-1)/2)
          begin
            if (r_Rx_Data == 1'b0)
              begin
                r_Clock_Count <= 0;  // reset counter, found the middle
                r_SM_Main     <= s_RX_DATA_BITS;
              end
            else
              r_SM_Main <= s_IDLE;
          end
        else
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= s_RX_START_BIT;
          end
      end // case: s_RX_START_BIT


    // Wait CLKS_PER_BIT-1 clock cycles to sample serial data
    s_RX_DATA_BITS :
      begin
        if (r_Clock_Count < CLKS_PER_BIT-1)
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= s_RX_DATA_BITS;
          end
        else
          begin
            r_Clock_Count          <= 0;
            r_Rx_Byte[r_Bit_Index] <= r_Rx_Data;

            // Check if we have received all bits
            if (r_Bit_Index < 7)
              begin
                r_Bit_Index <= r_Bit_Index + 1;
                r_SM_Main   <= s_RX_DATA_BITS;
              end
            else
              begin
                r_Bit_Index <= 0;
                r_SM_Main   <= s_RX_STOP_BIT;
              end
          end
      end // case: s_RX_DATA_BITS


    // Receive Stop bit.  Stop bit = 1
    s_RX_STOP_BIT :
      begin
        // Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
        if (r_Clock_Count < CLKS_PER_BIT-1)
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= s_RX_STOP_BIT;
          end
        else
          begin
            r_Rx_DV       <= 1'b1;
            r_Clock_Count <= 0;
            r_SM_Main     <= s_CLEANUP;
```

```
            end
          end // case: s_RX_STOP_BIT


        // Stay here 1 clock
        s_CLEANUP :
          begin
            r_SM_Main <= s_IDLE;
            r_Rx_DV   <= 1'b0;
          end


        default :
          r_SM_Main <= s_IDLE;

      endcase
    end

  assign o_Rx_DV   = r_Rx_DV;
  assign o_Rx_Byte = r_Rx_Byte;

endmodule // uart_rx

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/17/2019 03:26:39 PM
// Design Name:
// Module Name: uart_tx
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module uart_tx
#(parameter CLKS_PER_BIT = 1736,
parameter WIDTH = $clog2(CLKS_PER_BIT),
parameter s_IDLE         = 3'b000,
parameter s_TX_START_BIT = 3'b001,
parameter s_TX_DATA_BITS = 3'b010,
parameter s_TX_STOP_BIT  = 3'b011,
parameter s_CLEANUP      = 3'b100)
  (
   input       i_Clock,
   input       i_Tx_DV,
   input [7:0] i_Tx_Byte,
   output      o_Tx_Active,
   output reg  o_Tx_Serial,
   output      o_Tx_Done
   );


  reg [2:0]    r_SM_Main     = 0;
  reg [WIDTH:0]    r_Clock_Count = 0;
  reg [2:0]    r_Bit_Index   = 0;
  reg [7:0]    r_Tx_Data     = 0;
  reg          r_Tx_Done     = 0;
  reg          r_Tx_Active   = 0;

  always @(posedge i_Clock)
    begin

      case (r_SM_Main)
        s_IDLE :
          begin
            o_Tx_Serial   <= 1'b1;         // Drive Line High for Idle
            r_Tx_Done     <= 1'b0;
```

```
    r_Clock_Count <= 0;
    r_Bit_Index   <= 0;

    if (i_Tx_DV == 1'b1)
      begin
        r_Tx_Active <= 1'b1;
        r_Tx_Data   <= i_Tx_Byte;
        r_SM_Main   <= s_TX_START_BIT;
      end
    else
      r_SM_Main <= s_IDLE;
  end // case: s_IDLE


// Send out Start Bit. Start bit = 0
s_TX_START_BIT :
  begin
    o_Tx_Serial <= 1'b0;

    // Wait CLKS_PER_BIT-1 clock cycles for start bit to finish
    if (r_Clock_Count < CLKS_PER_BIT-1)
      begin
        r_Clock_Count <= r_Clock_Count + 1;
        r_SM_Main     <= s_TX_START_BIT;
      end
    else
      begin
        r_Clock_Count <= 0;
        r_SM_Main     <= s_TX_DATA_BITS;
      end
  end // case: s_TX_START_BIT


// Wait CLKS_PER_BIT-1 clock cycles for data bits to finish
s_TX_DATA_BITS :
  begin
    o_Tx_Serial <= r_Tx_Data[r_Bit_Index];

    if (r_Clock_Count < CLKS_PER_BIT-1)
      begin
        r_Clock_Count <= r_Clock_Count + 1;
        r_SM_Main     <= s_TX_DATA_BITS;
      end
    else
      begin
        r_Clock_Count <= 0;

        // Check if we have sent out all bits
        if (r_Bit_Index < 7)
          begin
            r_Bit_Index <= r_Bit_Index + 1;
            r_SM_Main   <= s_TX_DATA_BITS;
          end
        else
          begin
            r_Bit_Index <= 0;
            r_SM_Main   <= s_TX_STOP_BIT;
          end
      end
  end // case: s_TX_DATA_BITS


// Send out Stop bit.  Stop bit = 1
s_TX_STOP_BIT :
  begin
    o_Tx_Serial <= 1'b1;

    // Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
    if (r_Clock_Count < CLKS_PER_BIT-1)
      begin
        r_Clock_Count <= r_Clock_Count + 1;
        r_SM_Main     <= s_TX_STOP_BIT;
      end
    else
      begin
        r_Tx_Done     <= 1'b1;
        r_Clock_Count <= 0;
        r_SM_Main     <= s_CLEANUP;
```

```
              r_Tx_Active   <= 1'b0;
            end
          end // case: s_Tx_STOP_BIT


        // Stay here 1 clock
        s_CLEANUP :
          begin
            r_Tx_Done <= 1'b1;
            r_SM_Main <= s_IDLE;
          end


        default :
          r_SM_Main <= s_IDLE;

      endcase
    end

  assign o_Tx_Active = r_Tx_Active;
  assign o_Tx_Done   = r_Tx_Done;

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/06/2017 04:24:05 PM
// Design Name:
// Module Name: voting
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module voting(
    input [3:0]inputs,
    input clk,
    input [1:0]input_err_code,
    output v
);

    wire a,b,c,d;
    assign a = inputs[2];
    assign b = inputs[1];
    assign c = inputs[0];
    assign d = inputs[3];

    reg x,y,z;


assign v = (x & y) | (y & z) | (x & z); // majority vote

always @(posedge clk)
begin
    case(input_err_code)
    // no error so far
    2'b11:
      begin
        x <= a;
        y <= b;
        z <= c;
      end
    // c was a faulty unit
    2'b00:
      begin
        x <= a;
```

```
        y <= b;
        z <= d;
      end
    // b was a faulty module
    2'b01:
      begin
        x <= a;
        y <= d;
        z <= c;
      end
    // a was a faulty module
    2'b10:
      begin
        x <= d;
        y <= b;
        z <= c;
      end
    default:
      begin
        x <= a;
        y <= b;
        z <= c;
      end
    endcase
end

endmodule
```

APPENDIX B

HDL CONVERTER CODE

```
from __future__ import absolute_import
from __future__ import print_function
import sys
import re


class Node(object):
    """ Abstact class for every element in parser """

    def children(self):
        pass

    def show(self, buf=sys.stdout, offset=0, attrnames=False, showlineno=True):
        indent = 2
        lead = ' ' * offset
        buf.write(lead + self.__class__.__name__ + ': ')
        if self.attr_names:
            if attrnames:
                nvlist = [(n, getattr(self, n)) for n in self.attr_names]
                attrstr = ', '.join('%s=%s' % (n, v) for (n, v) in nvlist)
            else:
                vlist = [getattr(self, n) for n in self.attr_names]
                attrstr = ', '.join('%s' % v for v in vlist)
            buf.write(attrstr)
        if showlineno:
            buf.write(' (at %s)' % self.lineno)
        buf.write('\n')
        for c in self.children():
            c.show(buf, offset + indent, attrnames, showlineno)

    def __eq__(self, other):
        if type(self) != type(other):
            return False
        self_attrs = tuple([getattr(self, a) for a in self.attr_names])
        other_attrs = tuple([getattr(other, a) for a in other.attr_names])
        if self_attrs != other_attrs:
            return False
        other_children = other.children()
        for i, c in enumerate(self.children()):
            if c != other_children[i]:
                return False
        return True

    def __ne__(self, other):
        return not self.__eq__(other)

    def __hash__(self):
        s = hash(tuple([getattr(self, a) for a in self.attr_names]))
        c = hash(self.children())
        return hash((s, c))


# -----------------------------------------------------------------------------
class Source(Node):
    attr_names = ('name',)

    def __init__(self, name, description, lineno=0):
        self.lineno = lineno
        self.name = name
        self.description = description

    def children(self):
        nodelist = []
        if self.description:
            nodelist.append(self.description)
        return tuple(nodelist)
```

```
class Description(Node):
    attr_names = ()

    def __init__(self, definitions, lineno=0):
        self.lineno = lineno
        self.definitions = definitions

    def children(self):
        nodelist = []
        if self.definitions:
            nodelist.extend(self.definitions)
        return tuple(nodelist)


class ModuleDef(Node):
    attr_names = ('name',)

    def __init__(self, name, paramlist, portlist, items, default_nettype='wire', lineno=0):
        self.lineno = lineno
        self.name = name
        self.paramlist = paramlist
        self.portlist = portlist
        self.items = items
        self.default_nettype = default_nettype

    def children(self):
        nodelist = []
        if self.paramlist:
            nodelist.append(self.paramlist)
        if self.portlist:
            nodelist.append(self.portlist)
        if self.items:
            nodelist.extend(self.items)
        return tuple(nodelist)


class Paramlist(Node):
    attr_names = ()

    def __init__(self, params, lineno=0):
        self.lineno = lineno
        self.params = params

    def children(self):
        nodelist = []
        if self.params:
            nodelist.extend(self.params)
        return tuple(nodelist)


class Portlist(Node):
    attr_names = ()

    def __init__(self, ports, lineno=0):
        self.lineno = lineno
        self.ports = ports

    def children(self):
        nodelist = []
        if self.ports:
            nodelist.extend(self.ports)
        return tuple(nodelist)


class Port(Node):
    attr_names = ('name', 'type',)

    def __init__(self, name, width, type, lineno=0):
        self.lineno = lineno
        self.name = name
        self.width = width
        self.type = type

    def children(self):
        nodelist = []
        if self.width:
```

```
            nodelist.append(self.width)
        return tuple(nodelist)


class Width(Node):
    attr_names = ()

    def __init__(self, msb, lsb, lineno=0):
        self.lineno = lineno
        self.msb = msb
        self.lsb = lsb

    def children(self):
        nodelist = []
        if self.msb:
            nodelist.append(self.msb)
        if self.lsb:
            nodelist.append(self.lsb)
        return tuple(nodelist)


class Length(Width):
    pass


class Identifier(Node):
    attr_names = ('name',)

    def __init__(self, name, scope=None, lineno=0):
        self.lineno = lineno
        self.name = name
        self.scope = scope

    def children(self):
        nodelist = []
        if self.scope:
            nodelist.append(self.scope)
        return tuple(nodelist)

    def __repr__(self):
        if self.scope is None:
            return self.name
        return self.scope.__repr__() + '.' + self.name


class Value(Node):
    attr_names = ()

    def __init__(self, value, lineno=0):
        self.lineno = lineno
        self.value = value

    def children(self):
        nodelist = []
        if self.value:
            nodelist.append(self.value)
        return tuple(nodelist)


class Constant(Value):
    attr_names = ('value',)

    def __init__(self, value, lineno=0):
        self.lineno = lineno
        self.value = value

    def children(self):
        nodelist = []
        return tuple(nodelist)

    def __repr__(self):
        return str(self.value)


class IntConst(Constant):
    pass
```

```
class FloatConst(Constant):
    pass


class StringConst(Constant):
    pass


class Variable(Value):
    attr_names = ('name', 'signed')

    def __init__(self, name, width=None, signed=False, lineno=0):
        self.lineno = lineno
        self.name = name
        self.width = width
        self.signed = signed

    def children(self):
        nodelist = []
        if self.width:
            nodelist.append(self.width)
        return tuple(nodelist)


class Input(Variable):
    pass


class Output(Variable):
    pass


class Inout(Variable):
    pass


class Tri(Variable):
    pass


class Wire(Variable):
    pass


class Reg(Variable):
    pass


class WireArray(Variable):
    attr_names = ('name', 'signed')

    def __init__(self, name, width, length, signed=False, lineno=0):
        self.lineno = lineno
        self.name = name
        self.width = width
        self.length = length
        self.signed = signed

    def children(self):
        nodelist = []
        if self.width:
            nodelist.append(self.width)
        if self.length:
            nodelist.append(self.length)
        return tuple(nodelist)


class RegArray(Variable):
    attr_names = ('name', 'signed')

    def __init__(self, name, width, length, signed=False, lineno=0):
        self.lineno = lineno
        self.name = name
        self.width = width
        self.length = length
        self.signed = signed

    def children(self):
```

```
        nodelist = []
        if self.width:
            nodelist.append(self.width)
        if self.length:
            nodelist.append(self.length)
        return tuple(nodelist)


class Integer(Variable):
    pass


class Real(Variable):
    pass


class Genvar(Variable):
    pass


class Ioport(Node):
    attr_names = ()

    def __init__(self, first, second=None, lineno=0):
        self.lineno = lineno
        self.first = first
        self.second = second

    def children(self):
        nodelist = []
        if self.first:
            nodelist.append(self.first)
        if self.second:
            nodelist.append(self.second)
        return tuple(nodelist)


class Parameter(Node):
    attr_names = ('name', 'signed')

    def __init__(self, name, value, width=None, signed=False, lineno=0):
        self.lineno = lineno
        self.name = name
        self.value = value
        self.width = width
        self.signed = signed

    def children(self):
        nodelist = []
        if self.value:
            nodelist.append(self.value)
        if self.width:
            nodelist.append(self.width)
        return tuple(nodelist)


class Localparam(Parameter):
    pass


class Supply(Parameter):
    pass


class Decl(Node):
    attr_names = ()

    def __init__(self, list, lineno=0):
        self.lineno = lineno
        self.list = list

    def children(self):
        nodelist = []
        if self.list:
            nodelist.extend(self.list)
        return tuple(nodelist)
```

```
class Concat(Node):
    attr_names = ()

    def __init__(self, list, lineno=0):
        self.lineno = lineno
        self.list = list

    def children(self):
        nodelist = []
        if self.list:
            nodelist.extend(self.list)
        return tuple(nodelist)


class LConcat(Concat):
    pass


class Repeat(Node):
    attr_names = ()

    def __init__(self, value, times, lineno=0):
        self.lineno = lineno
        self.value = value
        self.times = times

    def children(self):
        nodelist = []
        if self.value:
            nodelist.append(self.value)
        if self.times:
            nodelist.append(self.times)
        return tuple(nodelist)


class Partselect(Node):
    attr_names = ()

    def __init__(self, var, msb, lsb, lineno=0):
        self.lineno = lineno
        self.var = var
        self.msb = msb
        self.lsb = lsb

    def children(self):
        nodelist = []
        if self.var:
            nodelist.append(self.var)
        if self.msb:
            nodelist.append(self.msb)
        if self.lsb:
            nodelist.append(self.lsb)
        return tuple(nodelist)


class Pointer(Node):
    attr_names = ()

    def __init__(self, var, ptr, lineno=0):
        self.lineno = lineno
        self.var = var
        self.ptr = ptr

    def children(self):
        nodelist = []
        if self.var:
            nodelist.append(self.var)
        if self.ptr:
            nodelist.append(self.ptr)
        return tuple(nodelist)


class Lvalue(Node):
    attr_names = ()

    def __init__(self, var, lineno=0):
        self.lineno = lineno
        self.var = var
```

```
    def children(self):
        nodelist = []
        if self.var:
            nodelist.append(self.var)
        return tuple(nodelist)


class Rvalue(Node):
    attr_names = ()

    def __init__(self, var, lineno=0):
        self.lineno = lineno
        self.var = var

    def children(self):
        nodelist = []
        if self.var:
            nodelist.append(self.var)
        return tuple(nodelist)


# ------------------------------------------------------------------------------
class Operator(Node):
    attr_names = ()

    def __init__(self, left, right, lineno=0):
        self.lineno = lineno
        self.left = left
        self.right = right

    def children(self):
        nodelist = []
        if self.left:
            nodelist.append(self.left)
        if self.right:
            nodelist.append(self.right)
        return tuple(nodelist)

    def __repr__(self):
        ret = '(' + self.__class__.__name__
        for c in self.children():
            ret += '␣' + c.__repr__()
        ret += ')'
        return ret


class UnaryOperator(Operator):
    attr_names = ()

    def __init__(self, right, lineno=0):
        self.lineno = lineno
        self.right = right

    def children(self):
        nodelist = []
        if self.right:
            nodelist.append(self.right)
        return tuple(nodelist)


# Level 1 (Highest Priority)
class Uplus(UnaryOperator):
    pass


class Uminus(UnaryOperator):
    pass


class Ulnot(UnaryOperator):
    pass


class Unot(UnaryOperator):
    pass
```

```
class Uand(UnaryOperator):
    pass


class Unand(UnaryOperator):
    pass


class Uor(UnaryOperator):
    pass


class Unor(UnaryOperator):
    pass


class Uxor(UnaryOperator):
    pass


class Uxnor(UnaryOperator):
    pass


# Level 2
class Power(Operator):
    pass


class Times(Operator):
    pass


class Divide(Operator):
    pass


class Mod(Operator):
    pass


# Level 3
class Plus(Operator):
    pass


class Minus(Operator):
    pass


# Level 4
class Sll(Operator):
    pass


class Srl(Operator):
    pass


class Sra(Operator):
    pass


# Level 5
class LessThan(Operator):
    pass


class GreaterThan(Operator):
    pass


class LessEq(Operator):
    pass


class GreaterEq(Operator):
    pass
```

```
# Level 6
class Eq(Operator):
    pass


class NotEq(Operator):
    pass


class Eql(Operator):
    pass  # ===


class NotEql(Operator):
    pass  # !==


# Level 7
class And(Operator):
    pass


class Xor(Operator):
    pass


class Xnor(Operator):
    pass


# Level 8
class Or(Operator):
    pass


# Level 9
class Land(Operator):
    pass


# Level 10
class Lor(Operator):
    pass


# Level 11
class Cond(Operator):
    attr_names = ()

    def __init__(self, cond, true_value, false_value, lineno=0):
        self.lineno = lineno
        self.cond = cond
        self.true_value = true_value
        self.false_value = false_value

    def children(self):
        nodelist = []
        if self.cond:
            nodelist.append(self.cond)
        if self.true_value:
            nodelist.append(self.true_value)
        if self.false_value:
            nodelist.append(self.false_value)
        return tuple(nodelist)


class Assign(Node):
    attr_names = ()

    def __init__(self, left, right, ldelay=None, rdelay=None, lineno=0):
        self.lineno = lineno
        self.left = left
        self.right = right
        self.ldelay = ldelay
        self.rdelay = rdelay
```

```
    def children(self):
        nodelist = []
        if self.left:
            nodelist.append(self.left)
        if self.right:
            nodelist.append(self.right)
        if self.ldelay:
            nodelist.append(self.ldelay)
        if self.rdelay:
            nodelist.append(self.rdelay)
        return tuple(nodelist)


class Always(Node):
    attr_names = ()

    def __init__(self, sens_list, statement, lineno=0):
        self.lineno = lineno
        self.sens_list = sens_list
        self.statement = statement

    def children(self):
        nodelist = []
        if self.sens_list:
            nodelist.append(self.sens_list)
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class AlwaysFF(Always):
    pass


class AlwaysComb(Always):
    pass


class AlwaysLatch(Always):
    pass


class SensList(Node):
    attr_names = ()

    def __init__(self, list, lineno=0):
        self.lineno = lineno
        self.list = list

    def children(self):
        nodelist = []
        if self.list:
            nodelist.extend(self.list)
        return tuple(nodelist)


class Sens(Node):
    attr_names = ('type',)

    def __init__(self, sig, type='posedge', lineno=0):
        self.lineno = lineno
        self.sig = sig
        self.type = type  # 'posedge', 'negedge', 'level', 'all' (*)

    def children(self):
        nodelist = []
        if self.sig:
            nodelist.append(self.sig)
        return tuple(nodelist)


class Substitution(Node):
    attr_names = ()

    def __init__(self, left, right, ldelay=None, rdelay=None, lineno=0):
        self.lineno = lineno
        self.left = left
        self.right = right
```

```
        self.ldelay = ldelay
        self.rdelay = rdelay

    def children(self):
        nodelist = []
        if self.left:
            nodelist.append(self.left)
        if self.right:
            nodelist.append(self.right)
        if self.ldelay:
            nodelist.append(self.ldelay)
        if self.rdelay:
            nodelist.append(self.rdelay)
        return tuple(nodelist)


class BlockingSubstitution(Substitution):
    pass


class NonblockingSubstitution(Substitution):
    pass


class IfStatement(Node):
    attr_names = ()

    def __init__(self, cond, true_statement, false_statement, lineno=0):
        self.lineno = lineno
        self.cond = cond
        self.true_statement = true_statement
        self.false_statement = false_statement

    def children(self):
        nodelist = []
        if self.cond:
            nodelist.append(self.cond)
        if self.true_statement:
            nodelist.append(self.true_statement)
        if self.false_statement:
            nodelist.append(self.false_statement)
        return tuple(nodelist)


class ForStatement(Node):
    attr_names = ()

    def __init__(self, pre, cond, post, statement, lineno=0):
        self.lineno = lineno
        self.pre = pre
        self.cond = cond
        self.post = post
        self.statement = statement

    def children(self):
        nodelist = []
        if self.pre:
            nodelist.append(self.pre)
        if self.cond:
            nodelist.append(self.cond)
        if self.post:
            nodelist.append(self.post)
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class WhileStatement(Node):
    attr_names = ()

    def __init__(self, cond, statement, lineno=0):
        self.lineno = lineno
        self.cond = cond
        self.statement = statement

    def children(self):
        nodelist = []
        if self.cond:
```

```
            nodelist.append(self.cond)
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class CaseStatement(Node):
    attr_names = ()

    def __init__(self, comp, caselist, lineno=0):
        self.lineno = lineno
        self.comp = comp
        self.caselist = caselist

    def children(self):
        nodelist = []
        if self.comp:
            nodelist.append(self.comp)
        if self.caselist:
            nodelist.extend(self.caselist)
        return tuple(nodelist)


class CasexStatement(CaseStatement):
    pass


class UniqueCaseStatement(CaseStatement):
    pass


class Case(Node):
    attr_names = ()

    def __init__(self, cond, statement, lineno=0):
        self.lineno = lineno
        self.cond = cond
        self.statement = statement

    def children(self):
        nodelist = []
        if self.cond:
            nodelist.extend(self.cond)
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class Block(Node):
    attr_names = ('scope',)

    def __init__(self, statements, scope=None, lineno=0):
        self.lineno = lineno
        self.statements = statements
        self.scope = scope

    def children(self):
        nodelist = []
        if self.statements:
            nodelist.extend(self.statements)
        return tuple(nodelist)


class Initial(Node):
    attr_names = ()

    def __init__(self, statement, lineno=0):
        self.lineno = lineno
        self.statement = statement

    def children(self):
        nodelist = []
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class EventStatement(Node):
```

```
    attr_names = ()

    def __init__(self, senslist, lineno=0):
        self.lineno = lineno
        self.senslist = senslist

    def children(self):
        nodelist = []
        if self.senslist:
            nodelist.append(self.senslist)
        return tuple(nodelist)


class WaitStatement(Node):
    attr_names = ()

    def __init__(self, cond, statement, lineno=0):
        self.lineno = lineno
        self.cond = cond
        self.statement = statement

    def children(self):
        nodelist = []
        if self.cond:
            nodelist.append(self.cond)
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class ForeverStatement(Node):
    attr_names = ()

    def __init__(self, statement, lineno=0):
        self.lineno = lineno
        self.statement = statement

    def children(self):
        nodelist = []
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class DelayStatement(Node):
    attr_names = ()

    def __init__(self, delay, lineno=0):
        self.lineno = lineno
        self.delay = delay

    def children(self):
        nodelist = []
        if self.delay:
            nodelist.append(self.delay)
        return tuple(nodelist)


class InstanceList(Node):
    attr_names = ('module',)

    def __init__(self, module, parameterlist, instances, lineno=0):
        self.lineno = lineno
        self.module = module
        self.parameterlist = parameterlist
        self.instances = instances

    def children(self):
        nodelist = []
        if self.parameterlist:
            nodelist.extend(self.parameterlist)
        if self.instances:
            nodelist.extend(self.instances)
        return tuple(nodelist)


class Instance(Node):
    attr_names = ('name', 'module')
```

```
    def __init__(self, module, name, portlist, parameterlist, array=None, lineno=0):
        self.lineno = lineno
        self.module = module
        self.name = name
        self.portlist = portlist
        self.parameterlist = parameterlist
        self.array = array

    def children(self):
        nodelist = []
        if self.array:
            nodelist.append(self.array)
        if self.parameterlist:
            nodelist.extend(self.parameterlist)
        if self.portlist:
            nodelist.extend(self.portlist)
        return tuple(nodelist)


class ParamArg(Node):
    attr_names = ('paramname',)

    def __init__(self, paramname, argname, lineno=0):
        self.lineno = lineno
        self.paramname = paramname
        self.argname = argname

    def children(self):
        nodelist = []
        if self.argname:
            nodelist.append(self.argname)
        return tuple(nodelist)


class PortArg(Node):
    attr_names = ('portname',)

    def __init__(self, portname, argname, lineno=0):
        self.lineno = lineno
        self.portname = portname
        self.argname = argname

    def children(self):
        nodelist = []
        if self.argname:
            nodelist.append(self.argname)
        return tuple(nodelist)


class Function(Node):
    attr_names = ('name',)

    def __init__(self, name, retwidth, statement, lineno=0):
        self.lineno = lineno
        self.name = name
        self.retwidth = retwidth
        self.statement = statement

    def children(self):
        nodelist = []
        if self.retwidth:
            nodelist.append(self.retwidth)
        if self.statement:
            nodelist.extend(self.statement)
        return tuple(nodelist)

    def __repr__(self):
        return self.name.__repr__()


class FunctionCall(Node):
    attr_names = ()

    def __init__(self, name, args, lineno=0):
        self.lineno = lineno
        self.name = name
        self.args = args
```

```
    def children(self):
        nodelist = []
        if self.name:
            nodelist.append(self.name)
        if self.args:
            nodelist.extend(self.args)
        return tuple(nodelist)

    def __repr__(self):
        return self.name.__repr__()


class Task(Node):
    attr_names = ('name',)

    def __init__(self, name, statement, lineno=0):
        self.lineno = lineno
        self.name = name
        self.statement = statement

    def children(self):
        nodelist = []
        if self.statement:
            nodelist.extend(self.statement)
        return tuple(nodelist)


class TaskCall(Node):
    attr_names = ()

    def __init__(self, name, args, lineno=0):
        self.lineno = lineno
        self.name = name
        self.args = args

    def children(self):
        nodelist = []
        if self.name:
            nodelist.append(self.name)
        if self.args:
            nodelist.extend(self.args)
        return tuple(nodelist)


class GenerateStatement(Node):
    attr_names = ()

    def __init__(self, items, lineno=0):
        self.lineno = lineno
        self.items = items

    def children(self):
        nodelist = []
        if self.items:
            nodelist.extend(self.items)
        return tuple(nodelist)


class SystemCall(Node):
    attr_names = ('syscall',)

    def __init__(self, syscall, args, lineno=0):
        self.lineno = lineno
        self.syscall = syscall
        self.args = args

    def children(self):
        nodelist = []
        if self.args:
            nodelist.extend(self.args)
        return tuple(nodelist)

    def __repr__(self):
        ret = []
        ret.append('(')
        ret.append('$')
        ret.append(self.syscall)
```

```
        for a in self.args:
            ret.append('␣')
            ret.append(str(a))
        ret.append(')')
        return ''.join(ret)


class IdentifierScopeLabel(Node):
    attr_names = ('name', 'loop')

    def __init__(self, name, loop=None, lineno=0):
        self.lineno = lineno
        self.name = name
        self.loop = loop

    def children(self):
        nodelist = []
        return tuple(nodelist)


class IdentifierScope(Node):
    attr_names = ()

    def __init__(self, labellist, lineno=0):
        self.lineno = lineno
        self.labellist = labellist

    def children(self):
        nodelist = []
        if self.labellist:
            nodelist.extend(self.labellist)
        return tuple(nodelist)


class Pragma(Node):
    attr_names = ()

    def __init__(self, entry, lineno=0):
        self.lineno = lineno
        self.entry = entry

    def children(self):
        nodelist = []
        if self.entry:
            nodelist.append(self.entry)
        return tuple(nodelist)


class PragmaEntry(Node):
    attr_names = ('name', )

    def __init__(self, name, value=None, lineno=0):
        self.lineno = lineno
        self.name = name
        self.value = value

    def children(self):
        nodelist = []
        if self.value:
            nodelist.append(self.value)
        return tuple(nodelist)


class Disable(Node):
    attr_names = ('dest',)

    def __init__(self, dest, lineno=0):
        self.lineno = lineno
        self.dest = dest

    def children(self):
        nodelist = []
        return tuple(nodelist)


class ParallelBlock(Node):
    attr_names = ('scope',)
```

```
    def __init__(self, statements, scope=None, lineno=0):
        self.lineno = lineno
        self.statements = statements
        self.scope = scope

    def children(self):
        nodelist = []
        if self.statements:
            nodelist.extend(self.statements)
        return tuple(nodelist)


class SingleStatement(Node):
    attr_names = ()

    def __init__(self, statement, lineno=0):
        self.lineno = lineno
        self.statement = statement

    def children(self):
        nodelist = []
        if self.statement:
            nodelist.append(self.statement)
        return tuple(nodelist)


class EmbeddedCode(Node):
    attr_names = ('code',)

    def __init__(self, code, lineno=0):
        self.code = code

    def children(self):
        nodelist = []
        return tuple(nodelist)

from __future__ import absolute_import
from __future__ import print_function
import sys
import os
import math
import re
import functools
from jinja2 import Environment, FileSystemLoader

from pyverilog.vparser.ast import *
from pyverilog.utils.op2mark import op2mark
from pyverilog.utils.op2mark import op2order

DEFAULT_TEMPLATE_DIR = os.path.dirname(os.path.abspath(__file__)) + '/template/'

# -------------------------------------------------------------------------------
try:
    import textwrap
    indent = textwrap.indent
except:
    def indent(text, prefix, predicate=None):
        if predicate is None:
            def predicate(x): return x and not x.isspace()
        ret = []
        for line in text.split('\n'):
            if predicate(line):
                ret.append(prefix)
            ret.append(line)
            ret.append('\n')
        return ''.join(ret[:-1])


def indent_multiline_assign(text):
    ret = []
    texts = text.split('\n')
    if len(texts) <= 1:
        return text
    try:
        p = texts[0].index('=')
    except:
        return text
    ret.append(texts[0])
```

```
    ret.append('\n')
    ret.append(indent('\n'.join(texts[1:]), '␣' * (p + 2)))
    return ''.join(ret)

# --------------------------------------------------------------------------------


class ConvertVisitor(object):
    def visit(self, node):
        method = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        ret = []
        for c in node.children():
            ret.append(self.visit(c))
        return ''.join(ret)


def getfilename(node):
    return node.__class__.__name__.lower() + '.txt'


def escape(s):
    if s.startswith('\\'):
        return s + '␣'
    return s


def del_paren(s):
    if s.startswith('(') and s.endswith(')'):
        return s[1:-1]
    return s


def del_space(s):
    return s.replace('␣', '')


class ASTCodeGenerator(ConvertVisitor):
    def __init__(self, indentsize=2):
        self.env = Environment(loader=FileSystemLoader(DEFAULT_TEMPLATE_DIR))
        self.indent = functools.partial(indent, prefix='␣' * indentsize)
        self.template_cache = {}

    def get_template(self, filename):
        if filename in self.template_cache:
            return self.template_cache[filename]

        template = self.env.get_template(filename)
        self.template_cache[filename] = template
        return template

    def visit_Source(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'description': self.visit(node.description),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Description(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'definitions': [self.visit(definition) for definition in node.definitions],
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_ModuleDef(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        paramlist = self.indent(self.visit(node.paramlist)) if node.paramlist is not None else ''
        portlist = self.indent(self.visit(node.portlist)) if node.portlist is not None else ''
        template_dict = {
```

```
            'modulename': escape(node.name),
            'paramlist': paramlist,
            'portlist': portlist,
            'items': [self.indent(self.visit(item)) for item in node.items] if node.items else (),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Paramlist(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        params = [self.visit(param).replace(';', '') for param in node.params]
        template_dict = {
            'params': params,
            'len_params': len(params),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Portlist(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        ports = [self.visit(port) for port in node.ports]
        template_dict = {
            'ports': ports,
            'len_ports': len(ports),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Port(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Width(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'msb': del_space(del_paren(self.visit(node.msb))),
            'lsb': del_space(del_paren(self.visit(node.lsb))),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Length(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'msb': del_space(del_paren(self.visit(node.msb))),
            'lsb': del_space(del_paren(self.visit(node.lsb))),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Identifier(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'scope': '' if node.scope is None else self.visit(node.scope),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Value(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'value': node.value,
        }
        rslt = template.render(template_dict)
        return rslt
```

```
def visit_Constant(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'value': node.value,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_IntConst(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'value': node.value,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_FloatConst(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'value': node.value,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_StringConst(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'value': node.value,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Variable(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'name': escape(node.name),
        'width': '' if node.width is None else self.visit(node.width),
        'signed': node.signed,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Input(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'name': escape(node.name),
        'width': '' if node.width is None else self.visit(node.width),
        'signed': node.signed,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Output(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'name': escape(node.name),
        'width': '' if node.width is None else self.visit(node.width),
        'signed': node.signed,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Inout(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'name': escape(node.name),
        'width': '' if node.width is None else self.visit(node.width),
        'signed': node.signed,
    }
    rslt = template.render(template_dict)
```

```
        return rslt

    def visit_Tri(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'width': '' if node.width is None else self.visit(node.width),
            'signed': node.signed,
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Wire(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'width': '' if node.width is None else self.visit(node.width),
            'signed': node.signed,
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Reg(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'width': '' if node.width is None else self.visit(node.width),
            'signed': node.signed,
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_WireArray(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'width': '' if node.width is None else self.visit(node.width),
            'length': self.visit(node.length),
            'signed': node.signed,
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_RegArray(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'width': '' if node.width is None else self.visit(node.width),
            'length': self.visit(node.length),
            'signed': node.signed,
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Integer(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'signed': node.signed,
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Real(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
        }
        rslt = template.render(template_dict)
        return rslt
```

```
def visit_Genvar(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'name': escape(node.name),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Ioport(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'first': node.first.__class__.__name__.lower(),
        'second': '' if node.second is None else node.second.__class__.__name__.lower(),
        'name': escape(node.first.name),
        'width': '' if node.first.width is None else self.visit(node.first.width),
        'signed': node.first.signed or (node.second is not None and node.second.signed)
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Parameter(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    value = self.visit(node.value)
    template_dict = {
        'name': escape(node.name),
        'width': '' if node.width is None or (value.startswith('"') and value.endswith('"')) else self.visit(node.width),
        'value': value,
        'signed': node.signed,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Localparam(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    value = self.visit(node.value)
    template_dict = {
        'name': escape(node.name),
        'width': '' if node.width is None or (value.startswith('"') and value.endswith('"')) else self.visit(node.width),
        'value': value,
        'signed': node.signed,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Decl(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'items': [self.visit(item) for item in node.list],
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Concat(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    items = [del_paren(self.visit(item)) for item in node.list]
    template_dict = {
        'items': items,
        'len_items': len(items),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_LConcat(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    items = [del_paren(self.visit(item)) for item in node.list]
    template_dict = {
        'items': items,
        'len_items': len(items),
    }
    rslt = template.render(template_dict)
    return rslt
```

```
def visit_Repeat(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'value': del_paren(self.visit(node.value)),
        'times': del_paren(self.visit(node.times)),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Partselect(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'var': self.visit(node.var),
        'msb': del_space(del_paren(self.visit(node.msb))),
        'lsb': del_space(del_paren(self.visit(node.lsb))),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Pointer(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'var': self.visit(node.var),
        'ptr': del_paren(self.visit(node.ptr)),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Lvalue(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'var': del_paren(self.visit(node.var)),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Rvalue(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'var': del_paren(self.visit(node.var)),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Operator(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    order = op2order(node.__class__.__name__)
    lorder = op2order(node.left.__class__.__name__)
    rorder = op2order(node.right.__class__.__name__)
    left = self.visit(node.left)
    right = self.visit(node.right)
    if ((not isinstance(node.left, (Sll, Srl, Sra,
                                    LessThan, GreaterThan, LessEq, GreaterEq,
                                    Eq, NotEq, Eql, NotEql))) and
            (lorder is not None and lorder <= order)):
        left = del_paren(left)
    if ((not isinstance(node.right, (Sll, Srl, Sra,
                                     LessThan, GreaterThan, LessEq, GreaterEq,
                                     Eq, NotEq, Eql, NotEql))) and
            (rorder is not None and order > rorder)):
        right = del_paren(right)
    template_dict = {
        'left': left,
        'right': right,
        'op': op2mark(node.__class__.__name__),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_UnaryOperator(self, node):
    filename = getfilename(node)
```

```
        template = self.get_template(filename)
        right = self.visit(node.right)
        template_dict = {
            'right': right,
            'op': op2mark(node.__class__.__name__),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Uplus(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Uminus(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Ulnot(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Unot(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Uand(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Unand(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Uor(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Unor(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Uxor(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Uxnor(self, node):
        return self.visit_UnaryOperator(node)

    def visit_Power(self, node):
        return self.visit_Operator(node)

    def visit_Times(self, node):
        return self.visit_Operator(node)

    def visit_Divide(self, node):
        return self.visit_Operator(node)

    def visit_Mod(self, node):
        return self.visit_Operator(node)

    def visit_Plus(self, node):
        return self.visit_Operator(node)

    def visit_Minus(self, node):
        return self.visit_Operator(node)

    def visit_Sll(self, node):
        return self.visit_Operator(node)

    def visit_Srl(self, node):
        return self.visit_Operator(node)

    def visit_Sra(self, node):
        return self.visit_Operator(node)

    def visit_LessThan(self, node):
        return self.visit_Operator(node)

    def visit_GreaterThan(self, node):
        return self.visit_Operator(node)

    def visit_LessEq(self, node):
        return self.visit_Operator(node)

    def visit_GreaterEq(self, node):
        return self.visit_Operator(node)
```

```
def visit_Eq(self, node):
    return self.visit_Operator(node)

def visit_NotEq(self, node):
    return self.visit_Operator(node)

def visit_Eql(self, node):
    return self.visit_Operator(node)

def visit_NotEql(self, node):
    return self.visit_Operator(node)

def visit_And(self, node):
    return self.visit_Operator(node)

def visit_Xor(self, node):
    return self.visit_Operator(node)

def visit_Xnor(self, node):
    return self.visit_Operator(node)

def visit_Or(self, node):
    return self.visit_Operator(node)

def visit_Land(self, node):
    return self.visit_Operator(node)

def visit_Lor(self, node):
    return self.visit_Operator(node)

def visit_Cond(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    true_value = del_paren(self.visit(node.true_value))
    false_value = del_paren(self.visit(node.false_value))
    if isinstance(node.false_value, Cond):
        false_value = ''.join(['\n', false_value])
    template_dict = {
        'cond': del_paren(self.visit(node.cond)),
        'true_value': true_value,
        'false_value': false_value,
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Assign(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'left': self.visit(node.left),
        'right': self.visit(node.right),
    }
    rslt = template.render(template_dict)
    rslt = indent_multiline_assign(rslt)
    return rslt

def visit_Always(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'sens_list': self.visit(node.sens_list),
        'statement': self.visit(node.statement),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_SensList(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    items = [self.visit(item) for item in node.list]
    template_dict = {
        'items': items,
        'len_items': len(items),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Sens(self, node):
```

```
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'sig': '*' if node.type == 'all' else self.visit(node.sig),
            'type': node.type if node.type == 'posedge' or node.type == 'negedge' else ''
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Substitution(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'left': self.visit(node.left),
            'right': self.visit(node.right),
            'ldelay': '' if node.ldelay is None else self.visit(node.ldelay),
            'rdelay': '' if node.rdelay is None else self.visit(node.rdelay),
        }
        rslt = template.render(template_dict)
        rslt = indent_multiline_assign(rslt)
        return rslt

    def visit_BlockingSubstitution(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'left': self.visit(node.left),
            'right': self.visit(node.right),
            'ldelay': '' if node.ldelay is None else self.visit(node.ldelay),
            'rdelay': '' if node.rdelay is None else self.visit(node.rdelay),
        }
        rslt = template.render(template_dict)
        rslt = indent_multiline_assign(rslt)
        return rslt

    def visit_NonblockingSubstitution(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'left': self.visit(node.left),
            'right': self.visit(node.right),
            'ldelay': '' if node.ldelay is None else self.visit(node.ldelay),
            'rdelay': '' if node.rdelay is None else self.visit(node.rdelay),
        }
        rslt = template.render(template_dict)
        rslt = indent_multiline_assign(rslt)
        return rslt

    def visit_IfStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        true_statement = '' if node.true_statement is None else self.visit(node.true_statement)
        false_statement = '' if node.false_statement is None else self.visit(node.false_statement)
        template_dict = {
            'cond': del_paren(self.visit(node.cond)),
            'true_statement': true_statement,
            'false_statement': false_statement,
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_ForStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'pre': '' if node.pre is None else del_space(self.visit(node.pre)),
            'cond': '' if node.cond is None else del_space(del_paren(self.visit(node.cond))),
            'post': '' if node.post is None else del_space(self.visit(node.post).replace(';', '')),
            'statement': '' if node.statement is None else self.visit(node.statement),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_WhileStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'cond': '' if node.cond is None else del_paren(self.visit(node.cond)),
```

```
            'statement': '' if node.statement is None else self.visit(node.statement),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_CaseStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'comp': del_paren(self.visit(node.comp)),
            'caselist': [self.indent(self.visit(case)) for case in node.caselist],
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_CasexStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'comp': del_paren(self.visit(node.comp)),
            'caselist': [self.indent(self.visit(case)) for case in node.caselist],
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Case(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        condlist = ['default'] if node.cond is None else [
            del_paren(self.visit(c)) for c in node.cond]
        cond = []
        for c in condlist:
            cond.append(c)
            cond.append(',_')
        template_dict = {
            'cond': ''.join(cond[:-1]),
            'statement': self.visit(node.statement),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Block(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'scope': '' if node.scope is None else escape(node.scope),
            'statements': [self.indent(self.visit(statement)) for statement in node.statements],
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Initial(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'statement': self.visit(node.statement),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_EventStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'senslist': del_paren(self.visit(node.senslist)),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_WaitStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'cond': del_paren(self.visit(node.cond)),
            'statement': self.visit(node.statement) if node.statement else '',
        }
        rslt = template.render(template_dict)
        return rslt
```

```
def visit_ForeverStatement(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'statement': self.visit(node.statement),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_DelayStatement(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'delay': self.visit(node.delay),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_InstanceList(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    parameterlist = [self.indent(self.visit(param)) for param in node.parameterlist]
    instances = [self.visit(instance) for instance in node.instances]
    template_dict = {
        'module': escape(node.module),
        'parameterlist': parameterlist,
        'len_parameterlist': len(parameterlist),
        'instances': instances,
        'len_instances': len(instances),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Instance(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    array = '' if node.array is None else self.visit(node.array)
    portlist = [self.indent(self.visit(port)) for port in node.portlist]
    template_dict = {
        'name': escape(node.name),
        'array': array,
        'portlist': portlist,
        'len_portlist': len(portlist),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_ParamArg(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'paramname': '' if node.paramname is None else escape(node.paramname),
        'argname': '' if node.argname is None else del_paren(self.visit(node.argname)),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_PortArg(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'portname': '' if node.portname is None else escape(node.portname),
        'argname': '' if node.argname is None else del_paren(self.visit(node.argname)),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Function(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    statement = [self.indent(self.visit(s)) for s in node.statement]
    template_dict = {
        'name': escape(node.name),
        'retwidth': self.visit(node.retwidth),
        'statement': statement,
    }
    rslt = template.render(template_dict)
```

```
    return rslt

def visit_FunctionCall(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    args = [self.visit(arg) for arg in node.args]
    template_dict = {
        'name': self.visit(node.name),
        'args': args,
        'len_args': len(args),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_Task(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    statement = [self.indent(self.visit(s)) for s in node.statement]
    template_dict = {
        'name': escape(node.name),
        'statement': statement,
    }
    rslt = template.render(template_dict)
    return rslt

# def visit_TaskCall(self, node):
#     filename = getfilename(node)
#     template = self.get_template(filename)
#     args = [ self.visit(arg) for arg in node.args ]
#     template_dict = {
#         'name' : self.visit(node.name),
#         'args' : args,
#         'len_args' : len(args),
#         }
#     rslt = template.render(template_dict)
#     return rslt

def visit_GenerateStatement(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'items': [self.visit(item) for item in node.items]
    }
    rslt = template.render(template_dict)
    return rslt

def visit_SystemCall(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    args = [self.visit(arg) for arg in node.args]
    template_dict = {
        'syscall': escape(node.syscall),
        'args': args,
        'len_args': len(args),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_IdentifierScopeLabel(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    template_dict = {
        'name': escape(node.name),
        'loop': '' if node.loop is None else self.visit(node.loop),
    }
    rslt = template.render(template_dict)
    return rslt

def visit_IdentifierScope(self, node):
    filename = getfilename(node)
    template = self.get_template(filename)
    scopes = [self.visit(scope) for scope in node.labellist]
    template_dict = {
        'scopes': scopes,
    }
    rslt = template.render(template_dict)
    return rslt
```

```
    def visit_Pragma(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'entry': self.visit(node.entry),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_PragmaEntry(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.name),
            'value': '' if node.value is None else self.visit(node.value),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_Disable(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'name': escape(node.dest),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_ParallelBlock(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'scope': '' if node.scope is None else escape(node.scope),
            'statements': [self.indent(self.visit(statement)) for statement in node.statements],
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_SingleStatement(self, node):
        filename = getfilename(node)
        template = self.get_template(filename)
        template_dict = {
            'statement': self.visit(node.statement),
        }
        rslt = template.render(template_dict)
        return rslt

    def visit_EmbeddedCode(self, node):
        return node.code

from __future__ import absolute_import
from __future__ import print_function
import sys
import os
import re

from pyverilog.vparser.ply.lex import *


class VerilogLexer(object):
    """ Verilog HDL Lexical Analayzer """

    def __init__(self, error_func):
        self.filename = ''
        self.error_func = error_func
        self.directives = []
        self.default_nettype = 'wire'

    def build(self, **kwargs):
        self.lexer = lex(object=self, **kwargs)

    def input(self, data):
        self.lexer.input(data)

    def reset_lineno(self):
        self.lexer.lineno = 1

    def get_directives(self):
```

```
        return tuple(self.directives)

    def get_default_nettype(self):
        return self.default_nettype

    def token(self):
        return self.lexer.token()

    keywords = (
        'MODULE', 'ENDMODULE', 'BEGIN', 'END', 'GENERATE', 'ENDGENERATE', 'GENVAR',
        'FUNCTION', 'ENDFUNCTION', 'TASK', 'ENDTASK',
        'INPUT', 'INOUT', 'OUTPUT', 'TRI', 'REG', 'LOGIC', 'WIRE', 'INTEGER', 'REAL', 'SIGNED',
        'PARAMETER', 'LOCALPARAM', 'SUPPLY0', 'SUPPLY1',
        'ASSIGN', 'ALWAYS', 'ALWAYS_FF', 'ALWAYS_COMB', 'ALWAYS_LATCH', 'SENS_OR', 'POSEDGE', 'NEGEDGE', 'INITIAL',
        'IF', 'ELSE', 'FOR', 'WHILE', 'CASE', 'CASEX', 'UNIQUE', 'ENDCASE', 'DEFAULT',
        'WAIT', 'FOREVER', 'DISABLE', 'FORK', 'JOIN',
    )

    reserved = {}
    for keyword in keywords:
        if keyword == 'SENS_OR':
            reserved['or'] = keyword
        else:
            reserved[keyword.lower()] = keyword

    operators = (
        'PLUS', 'MINUS', 'POWER', 'TIMES', 'DIVIDE', 'MOD',
        'NOT', 'OR', 'NOR', 'AND', 'NAND', 'XOR', 'XNOR',
        'LOR', 'LAND', 'LNOT',
        'LSHIFTA', 'RSHIFTA', 'LSHIFT', 'RSHIFT',
        'LT', 'GT', 'LE', 'GE', 'EQ', 'NE', 'EQL', 'NEL',
        'COND',  # ?
        'EQUALS',
    )

    tokens = keywords + operators + (
        'ID',
        'AT', 'COMMA', 'COLON', 'SEMICOLON', 'DOT',
        'PLUSCOLON', 'MINUSCOLON',
        'FLOATNUMBER', 'STRING_LITERAL',
        'INTNUMBER_DEC', 'SIGNED_INTNUMBER_DEC',
        'INTNUMBER_HEX', 'SIGNED_INTNUMBER_HEX',
        'INTNUMBER_OCT', 'SIGNED_INTNUMBER_OCT',
        'INTNUMBER_BIN', 'SIGNED_INTNUMBER_BIN',
        'LPAREN', 'RPAREN', 'LBRACKET', 'RBRACKET', 'LBRACE', 'RBRACE',
        'DELAY', 'DOLLER',
    )

    skipped = (
        'COMMENTOUT', 'LINECOMMENT', 'DIRECTIVE',
    )

    # Ignore
    t_ignore = ' \t'

    # Directive
    directive = r"""\`.*?\n"""

    @TOKEN(directive)
    def t_DIRECTIVE(self, t):
        self.directives.append((self.lexer.lineno, t.value))
        t.lexer.lineno += t.value.count("\n")
        m = re.match("^`default_nettype\s+(.+)\n", t.value)
        if m:
            self.default_nettype = m.group(1)
        pass

    # Comment
    linecomment = r"""//.*?\n"""
    commentout = r"""/\*(.|\n)*?\*/"""

    @TOKEN(linecomment)
    def t_LINECOMMENT(self, t):
        t.lexer.lineno += t.value.count("\n")
        pass

    @TOKEN(commentout)
    def t_COMMENTOUT(self, t):
```

```
    t.lexer.lineno += t.value.count("\n")
    pass

# Operator
t_LOR = r'\|\|'
t_LAND = r'\&\&'

t_NOR = r'~\|'
t_NAND = r'~\&'
t_XNOR = r'~\^'
t_OR = r'\|'
t_AND = r'\&'
t_XOR = r'\^'
t_LNOT = r'!'
t_NOT = r'~'

t_LSHIFTA = r'<<<'
t_RSHIFTA = r'>>>'
t_LSHIFT = r'<<'
t_RSHIFT = r'>>'

t_EQL = r'==='
t_NEL = r'!=='
t_EQ = r'=='
t_NE = r'!='

t_LE = r'<='
t_GE = r'>='
t_LT = r'<'
t_GT = r'>'

t_POWER = r'\*\*'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_MOD = r'%'

t_COND = r'\?'
t_EQUALS = r'='

t_PLUSCOLON = r'\+:'
t_MINUSCOLON = r'-:'

t_AT = r'@'
t_COMMA = r','
t_SEMICOLON = r';'
t_COLON = r':'
t_DOT = r'\.'

t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_LBRACE = r'\{'
t_RBRACE = r'\}'

t_DELAY = r'\#'
t_DOLLER = r'\$'

bin_number = '[0-9]*\'[bB][0-1xXzZ?][0-1xXzZ?_]*'
signed_bin_number = '[0-9]*\'[sS][bB][0-1xXzZ?][0-1xXzZ?_]*'
octal_number = '[0-9]*\'[oO][0-7xXzZ?][0-7xXzZ?_]*'
signed_octal_number = '[0-9]*\'[sS][oO][0-7xXzZ?][0-7xXzZ?_]*'
hex_number = '[0-9]*\'[hH][0-9a-fA-FxXzZ?][0-9a-fA-FxXzZ?_]*'
signed_hex_number = '[0-9]*\'[sS][hH][0-9a-fA-FxXzZ?][0-9a-fA-FxXzZ?_]*'

decimal_number = '([0-9]*\'[dD][0-9xXzZ?][0-9xXzZ?_]*)|([0-9][0-9_]*)'
signed_decimal_number = '[0-9]*\'[sS][dD][0-9xXzZ?][0-9xXzZ?_]*'

exponent_part = r"""([eE][-+]?[0-9]+)"""
fractional_constant = r"""([0-9]*\.[0-9]+)|([0-9]+\.)"""
float_number = '((((' + fractional_constant + ')' + \
    exponent_part + '?)|([0-9]+' + exponent_part + ')))'

simple_escape = r"""([a-zA-Z\\?'"])"""
octal_escape = r"""([0-7]{1,3})"""
hex_escape = r"""(x[0-9a-fA-F]+)"""
```

```
    escape_sequence = r"""(\\("""" + simple_escape + '|' + octal_escape + '|' + hex_escape + '))'
    string_char = r"""([^"\\\n]|""" + escape_sequence + ')'
    string_literal = '"' + string_char + '*"'

    identifier = r"""(([a-zA-Z_])([a-zA-Z_0-9$])*)|((\\S)(\S)*)"""

    @TOKEN(string_literal)
    def t_STRING_LITERAL(self, t):
        return t

    @TOKEN(float_number)
    def t_FLOATNUMBER(self, t):
        return t

    @TOKEN(signed_bin_number)
    def t_SIGNED_INTNUMBER_BIN(self, t):
        return t

    @TOKEN(bin_number)
    def t_INTNUMBER_BIN(self, t):
        return t

    @TOKEN(signed_octal_number)
    def t_SIGNED_INTNUMBER_OCT(self, t):
        return t

    @TOKEN(octal_number)
    def t_INTNUMBER_OCT(self, t):
        return t

    @TOKEN(signed_hex_number)
    def t_SIGNED_INTNUMBER_HEX(self, t):
        return t

    @TOKEN(hex_number)
    def t_INTNUMBER_HEX(self, t):
        return t

    @TOKEN(signed_decimal_number)
    def t_SIGNED_INTNUMBER_DEC(self, t):
        return t

    @TOKEN(decimal_number)
    def t_INTNUMBER_DEC(self, t):
        return t

    @TOKEN(identifier)
    def t_ID(self, t):
        t.type = self.reserved.get(t.value, 'ID')
        return t

    def t_NEWLINE(self, t):
        r'\n+'
        t.lexer.lineno += t.value.count("\n")
        pass

    def t_error(self, t):
        msg = 'Illegal character %s' % repr(t.value[0])
        self._error(msg, t)

    def _error(self, msg, token):
        location = self._make_tok_location(token)
        self.error_func(msg, location[0], location[1])
        self.lexer.skip(1)

    def _find_tok_column(self, token):
        i = token.lexpos
        while i > 0:
            if self.lexer.lexdata[i] == '\n':
                break
            i -= 1
        return (token.lexpos - i) + 1

    def _make_tok_location(self, token):
        return (token.lineno, self._find_tok_column(token))


def dump_tokens(text):
```

```
    def my_error_func(msg, a, b):
        sys.write(msg + "\n")
        sys.exit()

    lexer = VerilogLexer(error_func=my_error_func)
    lexer.build()
    lexer.input(text)

    ret = []

    # Tokenize
    while True:
        tok = lexer.token()
        if not tok:
            break  # No more input
        ret.append("%s %s %d %s %d\n" %
                   (tok.value, tok.type, tok.lineno, lexer.filename, tok.lexpos))

    return ''.join(ret)

from __future__ import absolute_import
from __future__ import print_function
import sys
import os

from pyverilog.vparser.ply.yacc import yacc
from pyverilog.vparser.plyparser import PLYParser, Coord, ParseError
from pyverilog.vparser.preprocessor import VerilogPreprocessor
from pyverilog.vparser.lexer import VerilogLexer
from pyverilog.vparser.ast import *


class VerilogParser(PLYParser):
    'Verilog HDL Parser'

    # Expression Precedence
    # Reference: http://hp.vector.co.jp/authors/VA016670/verilog/index.html
    precedence = (
        # <- Weak
        ('left', 'LOR'),
        ('left', 'LAND'),
        ('left', 'OR'),
        ('left', 'AND', 'XOR', 'XNOR'),
        ('left', 'EQ', 'NE', 'EQL', 'NEL'),
        ('left', 'LT', 'GT', 'LE', 'GE'),
        ('left', 'LSHIFT', 'RSHIFT', 'LSHIFTA', 'RSHIFTA'),
        ('left', 'PLUS', 'MINUS'),
        ('left', 'TIMES', 'DIVIDE', 'MOD'),
        ('left', 'POWER'),
        ('right', 'UMINUS', 'UPLUS', 'ULNOT', 'UNOT',
         'UAND', 'UNAND', 'UOR', 'UNOR', 'UXOR', 'UXNOR'),
        # -> Strong
    )

    def __init__(self):
        self.lexer = VerilogLexer(error_func=self._lexer_error_func)
        self.lexer.build()

        self.tokens = self.lexer.tokens
        #self.parser = yacc(module=self)
        # Use this if you want to build the parser using LALR(1) instead of SLR
        self.parser = yacc(module=self, method="LALR")

    def _lexer_error_func(self, msg, line, column):
        self._parse_error(msg, self._coord(line, column))

    def get_directives(self):
        return self.lexer.get_directives()

    def get_default_nettype(self):
        return self.lexer.get_default_nettype()

    # Returns AST
    def parse(self, text, debug=0):
        return self.parser.parse(text, lexer=self.lexer, debug=debug)

    # ----------------------------------------------------------------------
    # Parse Rule Definition
```

```
# ------------------------------------------------------------------------
def p_source_text(self, p):
    'source_text : description'
    p[0] = Source(name='', description=p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_description(self, p):
    'description : definitions'
    p[0] = Description(definitions=p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_definitions(self, p):
    'definitions : definitions definition'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_definitions_one(self, p):
    'definitions : definition'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_definition(self, p):
    'definition : moduledef'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_definition_pragma(self, p):
    'definition : pragma'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_pragma_assign(self, p):
    'pragma : LPAREN TIMES ID EQUALS expression TIMES RPAREN'
    p[0] = Pragma(PragmaEntry(p[3], p[5], lineno=p.lineno(1)),
                  lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_pragma(self, p):
    'pragma : LPAREN TIMES ID TIMES RPAREN'
    p[0] = Pragma(PragmaEntry(p[3], lineno=p.lineno(1)),
                  lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_moduledef(self, p):
    'moduledef : MODULE modulename paramlist portlist items ENDMODULE'
    p[0] = ModuleDef(name=p[2], paramlist=p[3], portlist=p[4], items=p[5],
                     default_nettype=self.get_default_nettype(), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))
    p[0].end_lineno = p.lineno(6)

def p_modulename(self, p):
    'modulename : ID'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_modulename_or(self, p):
    'modulename : SENS_OR'  # or primitive
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_paramlist(self, p):
    'paramlist : DELAY LPAREN params RPAREN'
    p[0] = Paramlist(params=p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_paramlist_empty(self, p):
    'paramlist : empty'
    p[0] = Paramlist(params=())

def p_params(self, p):
    'params : params_begin param_end'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_params_begin(self, p):
    'params_begin : params_begin param'
```

```
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_params_begin_one(self, p):
    'params_begin : param'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_params_one(self, p):
    'params : param_end'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_param(self, p):
    'param : PARAMETER param_substitution_list COMMA'
    paramlist = [Parameter(rname, rvalue, lineno=p.lineno(2))
                 for rname, rvalue in p[2]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_signed(self, p):
    'param : PARAMETER SIGNED param_substitution_list COMMA'
    paramlist = [Parameter(rname, rvalue, signed=True, lineno=p.lineno(2))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_width(self, p):
    'param : PARAMETER width param_substitution_list COMMA'
    paramlist = [Parameter(rname, rvalue, p[2], lineno=p.lineno(3))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_signed_width(self, p):
    'param : PARAMETER SIGNED width param_substitution_list COMMA'
    paramlist = [Parameter(rname, rvalue, p[3], signed=True, lineno=p.lineno(3))
                 for rname, rvalue in p[4]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_integer(self, p):
    'param : PARAMETER INTEGER param_substitution_list COMMA'
    paramlist = [Parameter(rname, rvalue, lineno=p.lineno(3))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_end(self, p):
    'param_end : PARAMETER param_substitution_list'
    paramlist = [Parameter(rname, rvalue, lineno=p.lineno(2))
                 for rname, rvalue in p[2]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_end_signed(self, p):
    'param_end : PARAMETER SIGNED param_substitution_list'
    paramlist = [Parameter(rname, rvalue, signed=True, lineno=p.lineno(2))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_end_width(self, p):
    'param_end : PARAMETER width param_substitution_list'
    paramlist = [Parameter(rname, rvalue, p[2], lineno=p.lineno(3))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_end_signed_width(self, p):
    'param_end : PARAMETER SIGNED width param_substitution_list'
    paramlist = [Parameter(rname, rvalue, p[3], signed=True, lineno=p.lineno(3))
                 for rname, rvalue in p[4]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_end_integer(self, p):
    'param_end : PARAMETER INTEGER param_substitution_list'
```

```
    paramlist = [Parameter(rname, rvalue, lineno=p.lineno(3))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_portlist(self, p):
    'portlist : LPAREN ports RPAREN SEMICOLON'
    p[0] = Portlist(ports=p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_portlist_io(self, p):
    'portlist : LPAREN ioports RPAREN SEMICOLON'
    p[0] = Portlist(ports=p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_portlist_paren_empty(self, p):
    'portlist : LPAREN RPAREN SEMICOLON'
    p[0] = Portlist(ports=(), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_portlist_empty(self, p):
    'portlist : SEMICOLON'
    p[0] = Portlist(ports=(), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_ports(self, p):
    'ports : ports COMMA portname'
    wid = None
    port = Port(name=p[3], width=wid, type=None, lineno=p.lineno(1))
    p[0] = p[1] + (port,)
    p.set_lineno(0, p.lineno(1))

def p_ports_one(self, p):
    'ports : portname'
    wid = None
    port = Port(name=p[1], width=wid, type=None, lineno=p.lineno(1))
    p[0] = (port,)
    p.set_lineno(0, p.lineno(1))

def p_portname(self, p):
    'portname : ID'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtypes(self, p):
    'sigtypes : sigtypes sigtype'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_sigtypes_one(self, p):
    'sigtypes : sigtype'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_sigtype_input(self, p):
    'sigtype : INPUT'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_output(self, p):
    'sigtype : OUTPUT'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_inout(self, p):
    'sigtype : INOUT'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_tri(self, p):
    'sigtype : TRI'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_reg(self, p):
    'sigtype : REG'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))
```

```
def p_sigtype_logic(self, p):
    'sigtype : LOGIC'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_wire(self, p):
    'sigtype : WIRE'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_signed(self, p):
    'sigtype : SIGNED'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_supply0(self, p):
    'sigtype : SUPPLY0'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_sigtype_supply1(self, p):
    'sigtype : SUPPLY1'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_ioports(self, p):
    'ioports : ioports COMMA ioport'
    if isinstance(p[3], str):
        t = None
        for r in reversed(p[1]):
            if isinstance(r.first, Input):
                t = Ioport(Input(name=p[3], width=r.first.width, lineno=p.lineno(3)),
                           lineno=p.lineno(3))
                break
            if isinstance(r.first, Output) and r.second is None:
                t = Ioport(Output(name=p[3], width=r.first.width, lineno=p.lineno(3)),
                           lineno=p.lineno(3))
                break
            if isinstance(r.first, Output) and isinstance(r.second, Reg):
                t = Ioport(Output(name=p[3], width=r.first.width, lineno=p.lineno(3)),
                           Reg(name=p[3], width=r.first.width,
                               lineno=p.lineno(3)),
                           lineno=p.lineno(3))
                break
            if isinstance(r.first, Inout):
                t = Ioport(Inout(name=p[3], width=r.first.width, lineno=p.lineno(3)),
                           lineno=p.lineno(3))
                break
        p[0] = p[1] + (t,)
    else:
        p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_ioports_one(self, p):
    'ioports : ioport_head'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def create_ioport(self, sigtypes, name, width=None, lineno=0):
    self.typecheck_ioport(sigtypes)
    first = None
    second = None
    signed = False
    if 'signed' in sigtypes:
        signed = True
    if 'input' in sigtypes:
        first = Input(name=name, width=width, signed=signed, lineno=lineno)
    if 'output' in sigtypes:
        first = Output(name=name, width=width,
                       signed=signed, lineno=lineno)
    if 'inout' in sigtypes:
        first = Inout(name=name, width=width, signed=signed, lineno=lineno)
    if 'wire' in sigtypes:
        second = Wire(name=name, width=width, signed=signed, lineno=lineno)
    if 'reg' in sigtypes:
        second = Reg(name=name, width=width, signed=signed, lineno=lineno)
    if 'tri' in sigtypes:
```

```
        second = Tri(name=name, width=width, signed=signed, lineno=lineno)
    return Ioport(first, second, lineno=lineno)

def typecheck_ioport(self, sigtypes):
    if 'input' not in sigtypes and 'output' not in sigtypes and 'inout' not in sigtypes:
        raise ParseError("Syntax Error")
    if 'input' in sigtypes and 'output' in sigtypes:
        raise ParseError("Syntax Error")
    if 'inout' in sigtypes and 'output' in sigtypes:
        raise ParseError("Syntax Error")
    if 'inout' in sigtypes and 'input' in sigtypes:
        raise ParseError("Syntax Error")
    if 'input' in sigtypes and 'reg' in sigtypes:
        raise ParseError("Syntax Error")
    if 'inout' in sigtypes and 'reg' in sigtypes:
        raise ParseError("Syntax Error")
    if 'input' in sigtypes and 'tri' in sigtypes:
        raise ParseError("Syntax Error")
    if 'output' in sigtypes and 'tri' in sigtypes:
        raise ParseError("Syntax Error")

def p_ioport(self, p):
    'ioport : sigtypes portname'
    p[0] = self.create_ioport(p[1], p[2], lineno=p.lineno(2))
    p.set_lineno(0, p.lineno(1))

def p_ioport_width(self, p):
    'ioport : sigtypes width portname'
    p[0] = self.create_ioport(p[1], p[3], width=p[2], lineno=p.lineno(3))
    p.set_lineno(0, p.lineno(1))

def p_ioport_head(self, p):
    'ioport_head : sigtypes portname'
    p[0] = self.create_ioport(p[1], p[2], lineno=p.lineno(2))
    p.set_lineno(0, p.lineno(1))

def p_ioport_head_width(self, p):
    'ioport_head : sigtypes width portname'
    p[0] = self.create_ioport(p[1], p[3], width=p[2], lineno=p.lineno(3))
    p.set_lineno(0, p.lineno(1))

def p_ioport_portname(self, p):
    'ioport : portname'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_width(self, p):
    'width : LBRACKET expression COLON expression RBRACKET'
    p[0] = Width(p[2], p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_length(self, p):
    'length : LBRACKET expression COLON expression RBRACKET'
    p[0] = Length(p[2], p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_items(self, p):
    'items : items item'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_items_one(self, p):
    'items : item'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_items_empty(self, p):
    'items : empty'
    p[0] = ()

def p_item(self, p):
    """item : standard_item
    | generate
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_standard_item(self, p):
```

```
    """standard_item : decl
    | integerdecl
    | realdecl
    | declassign
    | parameterdecl
    | localparamdecl
    | genvardecl
    | assignment
    | always
    | always_ff
    | always_comb
    | always_latch
    | initial
    | instance
    | function
    | task
    | pragma
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# Signal Decl
def create_decl(self, sigtypes, name, width=None, length=None, lineno=0):
    self.typecheck_decl(sigtypes, length)
    decls = []
    signed = False
    if 'signed' in sigtypes:
        signed = True
    if 'input' in sigtypes:
        decls.append(Input(name=name, width=width,
                           signed=signed, lineno=lineno))
    if 'output' in sigtypes:
        decls.append(Output(name=name, width=width,
                            signed=signed, lineno=lineno))
    if 'inout' in sigtypes:
        decls.append(Inout(name=name, width=width,
                           signed=signed, lineno=lineno))
    if 'wire' in sigtypes:
        if length:
            decls.append(WireArray(name=name, width=width,
                                   signed=signed, length=length, lineno=lineno))
        else:
            decls.append(Wire(name=name, width=width,
                              signed=signed, lineno=lineno))
    if 'reg' in sigtypes:
        if length:
            decls.append(RegArray(name=name, width=width,
                                  signed=signed, length=length, lineno=lineno))
        else:
            decls.append(Reg(name=name, width=width,
                             signed=signed, lineno=lineno))
    if 'tri' in sigtypes:
        decls.append(Tri(name=name, width=width,
                         signed=signed, lineno=lineno))
    if 'supply0' in sigtypes:
        decls.append(Supply(name=name, value=IntConst('0', lineno=lineno),
                            width=width, signed=signed, lineno=lineno))
    if 'supply1' in sigtypes:
        decls.append(Supply(name=name, value=IntConst('1', lineno=lineno),
                            width=width, signed=signed, lineno=lineno))
    return decls

def typecheck_decl(self, sigtypes, length=None):
    if length and 'input' in sigtypes:
        raise ParseError("Syntax Error")
    if length and 'output' in sigtypes:
        raise ParseError("Syntax Error")
    if length and 'inout' in sigtypes:
        raise ParseError("Syntax Error")
    if len(sigtypes) == 1 and 'signed' in sigtypes:
        raise ParseError("Syntax Error")
    if 'input' in sigtypes and 'output' in sigtypes:
        raise ParseError("Syntax Error")
    if 'inout' in sigtypes and 'output' in sigtypes:
        raise ParseError("Syntax Error")
    if 'inout' in sigtypes and 'input' in sigtypes:
        raise ParseError("Syntax Error")
    if 'input' in sigtypes and 'reg' in sigtypes:
```

```
            raise ParseError("Syntax Error")
        if 'inout' in sigtypes and 'reg' in sigtypes:
            raise ParseError("Syntax Error")
        if 'input' in sigtypes and 'tri' in sigtypes:
            raise ParseError("Syntax Error")
        if 'output' in sigtypes and 'tri' in sigtypes:
            raise ParseError("Syntax Error")

    def p_decl(self, p):
        'decl : sigtypes declnamelist SEMICOLON'
        decllist = []
        for rname, rlength in p[2]:
            decllist.extend(self.create_decl(p[1], rname, length=rlength,
                                             lineno=p.lineno(2)))
        p[0] = Decl(tuple(decllist), lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_decl_width(self, p):
        'decl : sigtypes width declnamelist SEMICOLON'
        decllist = []
        for rname, rlength in p[3]:
            decllist.extend(self.create_decl(p[1], rname, width=p[2], length=rlength,
                                             lineno=p.lineno(3)))
        p[0] = Decl(tuple(decllist), lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_declnamelist(self, p):
        'declnamelist : declnamelist COMMA declname'
        p[0] = p[1] + (p[3],)
        p.set_lineno(0, p.lineno(1))

    def p_declnamelist_one(self, p):
        'declnamelist : declname'
        p[0] = (p[1],)
        p.set_lineno(0, p.lineno(1))

    def p_declname(self, p):
        'declname : ID'
        p[0] = (p[1], None)
        p.set_lineno(0, p.lineno(1))

    def p_declarray(self, p):
        'declname : ID length'
        p[0] = (p[1], p[2])
        p.set_lineno(0, p.lineno(1))

    # Decl and Assign
    def create_declassign(self, sigtypes, name, assign, width=None, lineno=0):
        self.typecheck_declassign(sigtypes)
        decls = []
        signed = False
        if 'signed' in sigtypes:
            signed = True
        if 'input' in sigtypes:
            decls.append(Input(name=name, width=width,
                               signed=signed, lineno=lineno))
        if 'output' in sigtypes:
            decls.append(Output(name=name, width=width,
                                signed=signed, lineno=lineno))
        if 'inout' in sigtypes:
            decls.append(Inout(name=name, width=width,
                               signed=signed, lineno=lineno))
        if 'wire' in sigtypes:
            decls.append(Wire(name=name, width=width,
                              signed=signed, lineno=lineno))
        if 'reg' in sigtypes:
            decls.append(Reg(name=name, width=width,
                             signed=signed, lineno=lineno))
        decls.append(assign)
        return decls

    def typecheck_declassign(self, sigtypes):
        if len(sigtypes) == 1 and 'signed' in sigtypes:
            raise ParseError("Syntax Error")
        if 'reg' not in sigtypes and 'wire' not in sigtypes:
            raise ParseError("Syntax Error")
        if 'input' in sigtypes and 'output' in sigtypes:
            raise ParseError("Syntax Error")
```

```
        if 'inout' in sigtypes and 'output' in sigtypes:
            raise ParseError("Syntax Error")
        if 'inout' in sigtypes and 'input' in sigtypes:
            raise ParseError("Syntax Error")
        if 'input' in sigtypes and 'reg' in sigtypes:
            raise ParseError("Syntax Error")
        if 'inout' in sigtypes and 'reg' in sigtypes:
            raise ParseError("Syntax Error")
        if 'supply0' in sigtypes and len(sigtypes) != 1:
            raise ParseError("Syntax Error")
        if 'supply1' in sigtypes and len(sigtypes) != 1:
            raise ParseError("Syntax Error")

    def p_declassign(self, p):
        'declassign : sigtypes declassign_element SEMICOLON'
        decllist = self.create_declassign(
            p[1], p[2][0], p[2][1], lineno=p.lineno(2))
        p[0] = Decl(decllist, lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_declassign_width(self, p):
        'declassign : sigtypes width declassign_element SEMICOLON'
        decllist = self.create_declassign(
            p[1], p[3][0], p[3][1], width=p[2], lineno=p.lineno(3))
        p[0] = Decl(tuple(decllist), lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_declassign_element(self, p):
        'declassign_element : ID EQUALS rvalue'
        assign = Assign(Lvalue(Identifier(p[1], lineno=p.lineno(1)), lineno=p.lineno(1)),
                        p[3], lineno=p.lineno(1))
        p[0] = (p[1], assign)
        p.set_lineno(0, p.lineno(1))

    def p_declassign_element_delay(self, p):
        'declassign_element : delays ID EQUALS delays rvalue'
        assign = Assign(Lvalue(Identifier(p[2], lineno=p.lineno(1)), lineno=p.lineno(2)),
                        p[5], p[1], p[4], lineno=p.lineno(2))
        p[0] = (p[1], assign)
        p.set_lineno(0, p.lineno(2))

    # Integer
    def p_integerdecl(self, p):
        'integerdecl : INTEGER integernamelist SEMICOLON'
        intlist = [Integer(r,
                           Width(msb=IntConst('31', lineno=p.lineno(2)),
                                 lsb=IntConst('0', lineno=p.lineno(2)),
                                 lineno=p.lineno(2)),
                           signed=True, lineno=p.lineno(2)) for r in p[2]]
        p[0] = Decl(tuple(intlist), lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_integerdecl_signed(self, p):
        'integerdecl : INTEGER SIGNED integernamelist SEMICOLON'
        intlist = [Integer(r,
                           Width(msb=IntConst('31', lineno=p.lineno(3)),
                                 lsb=IntConst('0', lineno=p.lineno(3)),
                                 lineno=p.lineno(3)),
                           signed=True, lineno=p.lineno(3)) for r in p[2]]
        p[0] = Decl(tuple(intlist), lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_integernamelist(self, p):
        'integernamelist : integernamelist COMMA integername'
        p[0] = p[1] + (p[3],)
        p.set_lineno(0, p.lineno(1))

    def p_integernamelist_one(self, p):
        'integernamelist : integername'
        p[0] = (p[1],)
        p.set_lineno(0, p.lineno(1))

    def p_integername(self, p):
        'integername : ID'
        p[0] = p[1]
        p.set_lineno(0, p.lineno(1))

    # Real
```

```
def p_realdecl(self, p):
    'realdecl : REAL realnamelist SEMICOLON'
    reallist = [Real(p[1],
                     Width(msb=IntConst('31', lineno=p.lineno(2)),
                           lsb=IntConst('0', lineno=p.lineno(2)),
                           lineno=p.lineno(2)),
                     lineno=p.lineno(2)) for r in p[2]]
    p[0] = Decl(tuple(reallist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_realnamelist(self, p):
    'realnamelist : realnamelist COMMA realname'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_realnamelist_one(self, p):
    'realnamelist : realname'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_realname(self, p):
    'realname : ID'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# Parameter
def p_parameterdecl(self, p):
    'parameterdecl : PARAMETER param_substitution_list SEMICOLON'
    paramlist = [Parameter(rname, rvalue, lineno=p.lineno(2))
                 for rname, rvalue in p[2]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_parameterdecl_signed(self, p):
    'parameterdecl : PARAMETER SIGNED param_substitution_list SEMICOLON'
    paramlist = [Parameter(rname, rvalue, signed=True, lineno=p.lineno(2))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_parameterdecl_width(self, p):
    'parameterdecl : PARAMETER width param_substitution_list SEMICOLON'
    paramlist = [Parameter(rname, rvalue, p[2], lineno=p.lineno(3))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_parameterdecl_signed_width(self, p):
    'parameterdecl : PARAMETER SIGNED width param_substitution_list SEMICOLON'
    paramlist = [Parameter(rname, rvalue, p[3], signed=True, lineno=p.lineno(3))
                 for rname, rvalue in p[4]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_parameterdecl_integer(self, p):
    'parameterdecl : PARAMETER INTEGER param_substitution_list SEMICOLON'
    paramlist = [Parameter(rname, rvalue, lineno=p.lineno(3))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_localparamdecl(self, p):
    'localparamdecl : LOCALPARAM param_substitution_list SEMICOLON'
    paramlist = [Localparam(rname, rvalue, lineno=p.lineno(2))
                 for rname, rvalue in p[2]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_localparamdecl_signed(self, p):
    'localparamdecl : LOCALPARAM SIGNED param_substitution_list SEMICOLON'
    paramlist = [Localparam(rname, rvalue, signed=True, lineno=p.lineno(2))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_localparamdecl_width(self, p):
    'localparamdecl : LOCALPARAM width param_substitution_list SEMICOLON'
    paramlist = [Localparam(rname, rvalue, p[2], lineno=p.lineno(3))
```

```
                for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_localparamdecl_signed_width(self, p):
    'localparamdecl : LOCALPARAM SIGNED width param_substitution_list SEMICOLON'
    paramlist = [Localparam(rname, rvalue, p[3], signed=True, lineno=p.lineno(3))
                 for rname, rvalue in p[4]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_localparamdecl_integer(self, p):
    'localparamdecl : LOCALPARAM INTEGER param_substitution_list SEMICOLON'
    paramlist = [Localparam(rname, rvalue, lineno=p.lineno(3))
                 for rname, rvalue in p[3]]
    p[0] = Decl(tuple(paramlist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_substitution_list(self, p):
    'param_substitution_list : param_substitution_list COMMA param_substitution'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_param_substitution_list_one(self, p):
    'param_substitution_list : param_substitution'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_param_substitution(self, p):
    'param_substitution : ID EQUALS rvalue'
    p[0] = (p[1], p[3])
    p.set_lineno(0, p.lineno(1))

def p_assignment(self, p):
    'assignment : ASSIGN lvalue EQUALS rvalue SEMICOLON'
    p[0] = Assign(p[2], p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_assignment_delay(self, p):
    'assignment : ASSIGN delays lvalue EQUALS delays rvalue SEMICOLON'
    p[0] = Assign(p[3], p[6], p[2], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# --------------------------------------------------------------------------
def p_lpartselect_lpointer(self, p):
    'lpartselect : pointer LBRACKET expression COLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lpartselect_lpointer_plus(self, p):
    'lpartselect : pointer LBRACKET expression PLUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Plus(p[3], p[5]), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lpartselect_lpointer_minus(self, p):
    'lpartselect : pointer LBRACKET expression MINUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Minus(p[3], p[5]), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lpartselect(self, p):
    'lpartselect : identifier LBRACKET expression COLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lpartselect_plus(self, p):
    'lpartselect : identifier LBRACKET expression PLUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Plus(p[3], p[5]), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lpartselect_minus(self, p):
    'lpartselect : identifier LBRACKET expression MINUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Minus(p[3], p[5]), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lpointer(self, p):
    'lpointer : pointer'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))
```

```
def p_lconcat(self, p):
    'lconcat : LBRACE lconcatlist RBRACE'
    p[0] = LConcat(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lconcatlist(self, p):
    'lconcatlist : lconcatlist COMMA lconcat_one'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_lconcatlist_one(self, p):
    'lconcatlist : lconcat_one'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_lconcat_one_identifier(self, p):
    'lconcat_one : identifier'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_lconcat_one_lpartselect(self, p):
    'lconcat_one : lpartselect'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_lconcat_one_lpointer(self, p):
    'lconcat_one : lpointer'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_lconcat_one_lconcat(self, p):
    'lconcat_one : lconcat'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_lvalue_partselect(self, p):
    'lvalue : lpartselect'
    p[0] = Lvalue(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lvalue_pointer(self, p):
    'lvalue : lpointer'
    p[0] = Lvalue(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lvalue_concat(self, p):
    'lvalue : lconcat'
    p[0] = Lvalue(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_lvalue_one(self, p):
    'lvalue : identifier'
    p[0] = Lvalue(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_rvalue(self, p):
    'rvalue : expression'
    p[0] = Rvalue(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 1 (Highest Priority)
def p_expression_uminus(self, p):
    'expression : MINUS expression %prec UMINUS'
    p[0] = Uminus(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_uplus(self, p):
    'expression : PLUS expression %prec UPLUS'
    p[0] = p[2]
    p.set_lineno(0, p.lineno(1))

def p_expression_ulnot(self, p):
    'expression : LNOT expression %prec ULNOT'
    p[0] = Ulnot(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))
```

```
def p_expression_unot(self, p):
    'expression : NOT expression %prec UNOT'
    p[0] = Unot(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_uand(self, p):
    'expression : AND expression %prec UAND'
    p[0] = Uand(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_unand(self, p):
    'expression : NAND expression %prec UNAND'
    p[0] = Unand(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_unor(self, p):
    'expression : NOR expression %prec UNOR'
    p[0] = Unor(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_uor(self, p):
    'expression : OR expression %prec UOR'
    p[0] = Uor(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_uxor(self, p):
    'expression : XOR expression %prec UXOR'
    p[0] = Uxor(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_uxnor(self, p):
    'expression : XNOR expression %prec UXNOR'
    p[0] = Uxnor(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 2
def p_expression_power(self, p):
    'expression : expression POWER expression'
    p[0] = Power(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 3
def p_expression_times(self, p):
    'expression : expression TIMES expression'
    p[0] = Times(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_div(self, p):
    'expression : expression DIVIDE expression'
    p[0] = Divide(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_mod(self, p):
    'expression : expression MOD expression'
    p[0] = Mod(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 4
def p_expression_plus(self, p):
    'expression : expression PLUS expression'
    p[0] = Plus(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_minus(self, p):
    'expression : expression MINUS expression'
    p[0] = Minus(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 5
def p_expression_sll(self, p):
    'expression : expression LSHIFT expression'
    p[0] = Sll(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))
```

```
def p_expression_srl(self, p):
    'expression : expression RSHIFT expression'
    p[0] = Srl(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_sla(self, p):
    'expression : expression LSHIFTA expression'
    p[0] = Sll(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_sra(self, p):
    'expression : expression RSHIFTA expression'
    p[0] = Sra(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 6
def p_expression_lessthan(self, p):
    'expression : expression LT expression'
    p[0] = LessThan(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_greaterthan(self, p):
    'expression : expression GT expression'
    p[0] = GreaterThan(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_lesseq(self, p):
    'expression : expression LE expression'
    p[0] = LessEq(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_greatereq(self, p):
    'expression : expression GE expression'
    p[0] = GreaterEq(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 7
def p_expression_eq(self, p):
    'expression : expression EQ expression'
    p[0] = Eq(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_noteq(self, p):
    'expression : expression NE expression'
    p[0] = NotEq(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_eql(self, p):
    'expression : expression EQL expression'
    p[0] = Eql(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_noteql(self, p):
    'expression : expression NEL expression'
    p[0] = NotEql(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 8
def p_expression_And(self, p):
    'expression : expression AND expression'
    p[0] = And(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_Xor(self, p):
    'expression : expression XOR expression'
    p[0] = Xor(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_expression_Xnor(self, p):
    'expression : expression XNOR expression'
    p[0] = Xnor(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 9
```

```
def p_expression_Or(self, p):
    'expression : expression OR expression'
    p[0] = Or(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 10
def p_expression_land(self, p):
    'expression : expression LAND expression'
    p[0] = Land(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 11
def p_expression_lor(self, p):
    'expression : expression LOR expression'
    p[0] = Lor(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Level 12
def p_expression_cond(self, p):
    'expression : expression COND expression COLON expression'
    p[0] = Cond(p[1], p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_expression_expr(self, p):
    'expression : LPAREN expression RPAREN'
    p[0] = p[2]
    p.set_lineno(0, p.lineno(2))

# ------------------------------------------------------------------------
def p_expression_concat(self, p):
    'expression : concat'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_expression_repeat(self, p):
    'expression : repeat'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_expression_partselect(self, p):
    'expression : partselect'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_expression_pointer(self, p):
    'expression : pointer'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_expression_functioncall(self, p):
    'expression : functioncall'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_expression_systemcall(self, p):
    'expression : systemcall'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_expression_id(self, p):
    'expression : identifier'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_expression_const(self, p):
    'expression : const_expression'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_concat(self, p):
    'concat : LBRACE concatlist RBRACE'
    p[0] = Concat(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))
```

```
def p_concatlist(self, p):
    'concatlist : concatlist COMMA expression'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_concatlist_one(self, p):
    'concatlist : expression'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_repeat(self, p):
    'repeat : LBRACE expression concat RBRACE'
    p[0] = Repeat(p[3], p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_partselect(self, p):
    'partselect : identifier LBRACKET expression COLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_partselect_plus(self, p):
    'partselect : identifier LBRACKET expression PLUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Plus(
        p[3], p[5], lineno=p.lineno(1)), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_partselect_minus(self, p):
    'partselect : identifier LBRACKET expression MINUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Minus(
        p[3], p[5], lineno=p.lineno(1)), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_partselect_pointer(self, p):
    'partselect : pointer LBRACKET expression COLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_partselect_pointer_plus(self, p):
    'partselect : pointer LBRACKET expression PLUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Plus(
        p[3], p[5], lineno=p.lineno(1)), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_partselect_pointer_minus(self, p):
    'partselect : pointer LBRACKET expression MINUSCOLON expression RBRACKET'
    p[0] = Partselect(p[1], p[3], Minus(
        p[3], p[5], lineno=p.lineno(1)), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_pointer(self, p):
    'pointer : identifier LBRACKET expression RBRACKET'
    p[0] = Pointer(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_pointer_pointer(self, p):
    'pointer : pointer LBRACKET expression RBRACKET'
    p[0] = Pointer(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------------
def p_const_expression_intnum(self, p):
    'const_expression : intnumber'
    p[0] = IntConst(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_const_expression_floatnum(self, p):
    'const_expression : floatnumber'
    p[0] = FloatConst(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_const_expression_stringliteral(self, p):
    'const_expression : stringliteral'
    p[0] = StringConst(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_floatnumber(self, p):
    'floatnumber : FLOATNUMBER'
    p[0] = p[1]
```

```
    p.set_lineno(0, p.lineno(1))

def p_intnumber(self, p):
    """intnumber : INTNUMBER_DEC
    | SIGNED_INTNUMBER_DEC
    | INTNUMBER_BIN
    | SIGNED_INTNUMBER_BIN
    | INTNUMBER_OCT
    | SIGNED_INTNUMBER_OCT
    | INTNUMBER_HEX
    | SIGNED_INTNUMBER_HEX
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# String Literal
def p_stringliteral(self, p):
    'stringliteral : STRING_LITERAL'
    p[0] = p[1][1:-1]  # strip \" and \"
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
# Always
def p_always(self, p):
    'always : ALWAYS senslist always_statement'
    p[0] = Always(p[2], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_always_ff(self, p):
    'always_ff : ALWAYS_FF senslist always_statement'
    p[0] = AlwaysFF(p[2], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_always_comb(self, p):
    'always_comb : ALWAYS_COMB senslist always_statement'
    p[0] = AlwaysComb(p[2], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_always_latch(self, p):
    'always_latch : ALWAYS_LATCH senslist always_statement'
    p[0] = AlwaysLatch(p[2], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_sens_egde_paren(self, p):
    'senslist : AT LPAREN edgesigs RPAREN'
    p[0] = SensList(p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_posedgesig(self, p):
    'edgesig : POSEDGE edgesig_base'
    p[0] = Sens(p[2], 'posedge', lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_negedgesig(self, p):
    'edgesig : NEGEDGE edgesig_base'
    p[0] = Sens(p[2], 'negedge', lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_edgesig_base_identifier(self, p):
    'edgesig_base : identifier'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_edgesig_base_pointer(self, p):
    'edgesig_base : pointer'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_edgesigs(self, p):
    'edgesigs : edgesigs SENS_OR edgesig'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_edgesigs_comma(self, p):
    'edgesigs : edgesigs COMMA edgesig'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))
```

```
def p_edgesigs_one(self, p):
    'edgesigs : edgesig'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_sens_empty(self, p):
    'senslist : empty'
    p[0] = SensList(
        (Sens(None, 'all', lineno=p.lineno(1)),), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_sens_level(self, p):
    'senslist : AT levelsig'
    p[0] = SensList((p[2],), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_sens_level_paren(self, p):
    'senslist : AT LPAREN levelsigs RPAREN'
    p[0] = SensList(p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_levelsig(self, p):
    'levelsig : levelsig_base'
    p[0] = Sens(p[1], 'level', lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_levelsig_base_identifier(self, p):
    'levelsig_base : identifier'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_levelsig_base_pointer(self, p):
    'levelsig_base : pointer'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_levelsig_base_partselect(self, p):
    'levelsig_base : partselect'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_levelsigs(self, p):
    'levelsigs : levelsigs SENS_OR levelsig'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_levelsigs_comma(self, p):
    'levelsigs : levelsigs COMMA levelsig'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_levelsigs_one(self, p):
    'levelsigs : levelsig'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_sens_all(self, p):
    'senslist : AT TIMES'
    p[0] = SensList(
        (Sens(None, 'all', lineno=p.lineno(1)),), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_sens_all_paren(self, p):
    'senslist : AT LPAREN TIMES RPAREN'
    p[0] = SensList(
        (Sens(None, 'all', lineno=p.lineno(1)),), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_basic_statement(self, p):
    """basic_statement : if_statement
    | case_statement
    | casex_statement
    | unique_case_statement
    | for_statement
    | while_statement
    | event_statement
    | wait_statement
```

```
    | forever_statement
    | block
    | namedblock
    | parallelblock
    | blocking_substitution
    | nonblocking_substitution
    | single_statement
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_always_statement(self, p):
    'always_statement : basic_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------------
def p_blocking_substitution(self, p):
    'blocking_substitution : delays lvalue EQUALS delays rvalue SEMICOLON'
    p[0] = BlockingSubstitution(p[2], p[5], p[1], p[4], lineno=p.lineno(2))
    p.set_lineno(0, p.lineno(2))

def p_blocking_substitution_base(self, p):
    'blocking_substitution_base : delays lvalue EQUALS delays rvalue'
    p[0] = BlockingSubstitution(p[2], p[5], p[1], p[4], lineno=p.lineno(2))
    p.set_lineno(0, p.lineno(2))

def p_nonblocking_substitution(self, p):
    'nonblocking_substitution : delays lvalue LE delays rvalue SEMICOLON'
    p[0] = NonblockingSubstitution(
        p[2], p[5], p[1], p[4], lineno=p.lineno(2))
    p.set_lineno(0, p.lineno(2))

# ----------------------------------------------------------------------------
def p_delays(self, p):
    'delays : DELAY LPAREN expression RPAREN'
    p[0] = DelayStatement(p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_delays_identifier(self, p):
    'delays : DELAY identifier'
    p[0] = DelayStatement(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_delays_intnumber(self, p):
    'delays : DELAY intnumber'
    p[0] = DelayStatement(
        IntConst(p[2], lineno=p.lineno(1)), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_delays_floatnumber(self, p):
    'delays : DELAY floatnumber'
    p[0] = DelayStatement(FloatConst(
        p[2], lineno=p.lineno(1)), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_delays_empty(self, p):
    'delays : empty'
    p[0] = None

# ----------------------------------------------------------------------------
def p_block(self, p):
    'block : BEGIN block_statements END'
    p[0] = Block(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_block_empty(self, p):
    'block : BEGIN END'
    p[0] = Block((), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_block_statements(self, p):
    'block_statements : block_statements block_statement'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_block_statements_one(self, p):
    'block_statements : block_statement'
```

```
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_block_statement(self, p):
    'block_statement : basic_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------------
def p_namedblock(self, p):
    'namedblock : BEGIN COLON ID namedblock_statements END'
    p[0] = Block(p[4], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_namedblock_empty(self, p):
    'namedblock : BEGIN COLON ID END'
    p[0] = Block((), p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_namedblock_statements(self, p):
    'namedblock_statements : namedblock_statements namedblock_statement'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_namedblock_statements_one(self, p):
    'namedblock_statements : namedblock_statement'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_namedblock_statement(self, p):
    """namedblock_statement : basic_statement
    | decl
    | integerdecl
    | realdecl
    | parameterdecl
    | localparamdecl
    """
    if isinstance(p[1], Decl):
        for r in p[1].list:
            if (not isinstance(r, Reg) and not isinstance(r, Wire)
                and not isinstance(r, Integer) and not isinstance(r, Real)
                    and not isinstance(r, Parameter) and not isinstance(r, Localparam)):
                raise ParseError("Syntax Error")
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------------
def p_parallelblock(self, p):
    'parallelblock : FORK block_statements JOIN'
    p[0] = ParallelBlock(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_parallelblock_empty(self, p):
    'parallelblock : FORK JOIN'
    p[0] = ParallelBlock((), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------------
def p_if_statement(self, p):
    'if_statement : IF LPAREN cond RPAREN true_statement ELSE else_statement'
    p[0] = IfStatement(p[3], p[5], p[7], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_if_statement_woelse(self, p):
    'if_statement : IF LPAREN cond RPAREN true_statement'
    p[0] = IfStatement(p[3], p[5], None, lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_if_statement_delay(self, p):
    'if_statement : delays IF LPAREN cond RPAREN true_statement ELSE else_statement'
    p[0] = IfStatement(p[4], p[6], p[8], lineno=p.lineno(2))
    p.set_lineno(0, p.lineno(2))

def p_if_statement_woelse_delay(self, p):
    'if_statement : delays IF LPAREN cond RPAREN true_statement'
    p[0] = IfStatement(p[4], p[6], None, lineno=p.lineno(2))
    p.set_lineno(0, p.lineno(2))
```

```
def p_cond(self, p):
    'cond : expression'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_ifcontent_statement(self, p):
    'ifcontent_statement : basic_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_true_statement(self, p):
    'true_statement : ifcontent_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_else_statement(self, p):
    'else_statement : ifcontent_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------
def p_for_statement(self, p):
    'for_statement : FOR LPAREN forpre forcond forpost RPAREN forcontent_statement'
    p[0] = ForStatement(p[3], p[4], p[5], p[7], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_forpre(self, p):
    'forpre : blocking_substitution'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_forpre_empty(self, p):
    'forpre : SEMICOLON'
    p[0] = None
    p.set_lineno(0, p.lineno(1))

def p_forcond(self, p):
    'forcond : cond SEMICOLON'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_forcond_empty(self, p):
    'forcond : SEMICOLON'
    p[0] = None
    p.set_lineno(0, p.lineno(1))

def p_forpost(self, p):
    'forpost : blocking_substitution_base'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_forpost_empty(self, p):
    'forpost : empty'
    p[0] = None

def p_forcontent_statement(self, p):
    'forcontent_statement : basic_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------
def p_while_statement(self, p):
    'while_statement : WHILE LPAREN cond RPAREN whilecontent_statement'
    p[0] = WhileStatement(p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_whilecontent_statement(self, p):
    'whilecontent_statement : basic_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------
def p_case_statement(self, p):
    'case_statement : CASE LPAREN case_comp RPAREN casecontent_statements ENDCASE'
    p[0] = CaseStatement(p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_casex_statement(self, p):
```

```
    'casex_statement : CASEX LPAREN case_comp RPAREN casecontent_statements ENDCASE'
    p[0] = CasexStatement(p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_unique_case_statement(self, p):
    'unique_case_statement : UNIQUE_CASE LPAREN case_comp RPAREN casecontent_statements ENDCASE'
    p[0] = UniqueCaseStatement(p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))


def p_case_comp(self, p):
    'case_comp : expression'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_casecontent_statements(self, p):
    'casecontent_statements : casecontent_statements casecontent_statement'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_casecontent_statements_one(self, p):
    'casecontent_statements : casecontent_statement'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_casecontent_statement(self, p):
    'casecontent_statement : casecontent_condition COLON basic_statement'
    p[0] = Case(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_casecontent_condition_single(self, p):
    'casecontent_condition : casecontent_condition COMMA expression'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_casecontent_condition_one(self, p):
    'casecontent_condition : expression'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_casecontent_statement_default(self, p):
    'casecontent_statement : DEFAULT COLON basic_statement'
    p[0] = Case(None, p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_initial(self, p):
    'initial : INITIAL initial_statement'
    p[0] = Initial(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_initial_statement(self, p):
    'initial_statement : basic_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_event_statement(self, p):
    'event_statement : senslist SEMICOLON'
    p[0] = EventStatement(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_wait_statement(self, p):
    'wait_statement : WAIT LPAREN cond RPAREN waitcontent_statement'
    p[0] = WaitStatement(p[3], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_waitcontent_statement(self, p):
    'waitcontent_statement : basic_statement'
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_waitcontent_statement_empty(self, p):
    'waitcontent_statement : SEMICOLON'
    p[0] = None
    p.set_lineno(0, p.lineno(1))
```

```
# ------------------------------------------------------------------------
def p_forever_statement(self, p):
    'forever_statement : FOREVER basic_statement'
    p[0] = ForeverStatement(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_instance(self, p):
    'instance : ID parameterlist instance_bodylist SEMICOLON'
    instancelist = []
    for instance_name, instance_ports, instance_array in p[3]:
        instancelist.append(Instance(p[1], instance_name, instance_ports,
                                     p[2], instance_array, lineno=p.lineno(1)))
    p[0] = InstanceList(p[1], p[2], tuple(
        instancelist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_or(self, p):
    'instance : SENS_OR parameterlist instance_bodylist SEMICOLON'
    instancelist = []
    for instance_name, instance_ports, instance_array in p[3]:
        instancelist.append(Instance(p[1], instance_name, instance_ports,
                                     p[2], instance_array, lineno=p.lineno(1)))
    p[0] = InstanceList(p[1], p[2], tuple(
        instancelist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_bodylist(self, p):
    'instance_bodylist : instance_bodylist COMMA instance_body'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_instance_bodylist_one(self, p):
    'instance_bodylist : instance_body'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_instance_body(self, p):
    'instance_body : ID LPAREN instance_ports RPAREN'
    p[0] = (p[1], p[3], None)
    p.set_lineno(0, p.lineno(1))

def p_instance_body_array(self, p):
    'instance_body : ID width LPAREN instance_ports RPAREN'
    p[0] = (p[1], p[4], p[2])
    p.set_lineno(0, p.lineno(1))

def p_instance_noname(self, p):
    'instance : ID instance_bodylist_noname SEMICOLON'
    instancelist = []
    for instance_name, instance_ports, instance_array in p[2]:
        instancelist.append(Instance(p[1], instance_name, instance_ports,
                                     (), instance_array, lineno=p.lineno(1)))
    p[0] = InstanceList(p[1], (), tuple(instancelist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_or_noname(self, p):
    'instance : SENS_OR instance_bodylist_noname SEMICOLON'
    instancelist = []
    for instance_name, instance_ports, instance_array in p[2]:
        instancelist.append(Instance(p[1], instance_name, instance_ports,
                                     (), instance_array, lineno=p.lineno(1)))
    p[0] = InstanceList(p[1], (), tuple(instancelist), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_bodylist_noname(self, p):
    'instance_bodylist_noname : instance_bodylist_noname COMMA instance_body_noname'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_instance_bodylist_one_noname(self, p):
    'instance_bodylist_noname : instance_body_noname'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_instance_body_noname(self, p):
    'instance_body_noname : LPAREN instance_ports RPAREN'
    p[0] = ('', p[2], None)
```

```
    p.set_lineno(0, p.lineno(1))

def p_parameterlist(self, p):
    'parameterlist : DELAY LPAREN param_args RPAREN'
    p[0] = p[3]
    p.set_lineno(0, p.lineno(1))

def p_parameterlist_noname(self, p):
    'parameterlist : DELAY LPAREN param_args_noname RPAREN'
    p[0] = p[3]
    p.set_lineno(0, p.lineno(1))

def p_parameterlist_empty(self, p):
    'parameterlist : empty'
    p[0] = ()

def p_param_args_noname(self, p):
    'param_args_noname : param_args_noname COMMA param_arg_noname'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_param_args_noname_one(self, p):
    'param_args_noname : param_arg_noname'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_param_args(self, p):
    'param_args : param_args COMMA param_arg'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_param_args_one(self, p):
    'param_args : param_arg'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_param_args_empty(self, p):
    'param_args : empty'
    p[0] = ()

def p_param_arg_noname_exp(self, p):
    'param_arg_noname : expression'
    p[0] = ParamArg(None, p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_param_arg_exp(self, p):
    'param_arg : DOT ID LPAREN expression RPAREN'
    p[0] = ParamArg(p[2], p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_ports(self, p):
    """instance_ports : instance_ports_list
    | instance_ports_arg
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_instance_ports_list(self, p):
    'instance_ports_list : instance_ports_list COMMA instance_port_list'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_instance_ports_list_one(self, p):
    'instance_ports_list : instance_port_list'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_instance_ports_list_empty(self, p):
    'instance_ports_list : empty'
    p[0] = ()
    p.set_lineno(0, p.lineno(1))

def p_instance_port_list(self, p):
    'instance_port_list : expression'
    p[0] = PortArg(None, p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_ports_arg(self, p):
```

```
    'instance_ports_arg_ : _instance_ports_arg _COMMA_ instance_port_arg'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_instance_ports_arg_one(self, p):
    'instance_ports_arg_ : _instance_port_arg'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_instance_port_arg(self, p):
    'instance_port_arg_ : _DOT_ID _LPAREN_ identifier _RPAREN'
    p[0] = PortArg(p[2], p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_port_arg_exp(self, p):
    'instance_port_arg_ : _DOT_ID _LPAREN_ expression _RPAREN'
    p[0] = PortArg(p[2], p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_instance_port_arg_none(self, p):
    'instance_port_arg_ : _DOT_ID _LPAREN_RPAREN'
    p[0] = PortArg(p[2], None, lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ----------------------------------------------------------------------
def p_genvardecl(self, p):
    'genvardecl_ : _GENVAR_ genvarlist _SEMICOLON'
    p[0] = Decl(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_genvarlist(self, p):
    'genvarlist_ : _genvarlist _COMMA_ genvar'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_genvarlist_one(self, p):
    'genvarlist_ : _genvar'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_genvar(self, p):
    'genvar_ : _ID'
    p[0] = Genvar(name=p[1],
                  width=Width(msb=IntConst('31', lineno=p.lineno(1)),
                              lsb=IntConst('0', lineno=p.lineno(1)),
                              lineno=p.lineno(1)),
                  lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_generate(self, p):
    'generate_ : _GENERATE_ generate_items _ENDGENERATE'
    p[0] = GenerateStatement(p[2], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_generate_items_empty(self, p):
    'generate_items_ : _empty'
    p[0] = ()
    p.set_lineno(0, p.lineno(1))

def p_generate_items(self, p):
    'generate_items_ : _generate_items _generate_item'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_generate_items_one(self, p):
    'generate_items_ : _generate_item'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_generate_item(self, p):
    """generate_item : standard_item
    | generate_if
    | generate_for
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_generate_block(self, p):
```

```
        'generate_block : BEGIN generate_items END'
        p[0] = Block(p[2], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_generate_named_block(self, p):
        'generate_block : BEGIN COLON ID generate_items END'
        p[0] = Block(p[4], p[3], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_generate_if(self, p):
        'generate_if : IF LPAREN cond RPAREN gif_true_item ELSE gif_false_item'
        p[0] = IfStatement(p[3], p[5], p[7], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_generate_if_woelse(self, p):
        'generate_if : IF LPAREN cond RPAREN gif_true_item'
        p[0] = IfStatement(p[3], p[5], None, lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_generate_if_true_item(self, p):
        """gif_true_item : generate_item
        | generate_block
        """
        p[0] = p[1]
        p.set_lineno(0, p.lineno(1))

    def p_generate_if_false_item(self, p):
        """gif_false_item : generate_item
        | generate_block
        """
        p[0] = p[1]
        p.set_lineno(0, p.lineno(1))

    def p_generate_for(self, p):
        'generate_for : FOR LPAREN forpre forcond forpost RPAREN generate_forcontent'
        p[0] = ForStatement(p[3], p[4], p[5], p[7], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_generate_forcontent(self, p):
        """generate_forcontent : generate_item
        | generate_block
        """
        p[0] = p[1]
        p.set_lineno(0, p.lineno(1))

    # ----------------------------------------------------------------------
    def p_systemcall_noargs(self, p):
        'systemcall : DOLLER_ID'
        p[0] = SystemCall(p[2], (), lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_systemcall(self, p):
        'systemcall : DOLLER_ID LPAREN sysargs RPAREN'
        p[0] = SystemCall(p[2], p[4], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_systemcall_signed(self, p):  # for $signed system task
        'systemcall : DOLLER_SIGNED LPAREN sysargs RPAREN'
        p[0] = SystemCall(p[2], p[4], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_sysargs(self, p):
        'sysargs : sysargs COMMA sysarg'
        p[0] = p[1] + (p[3],)
        p.set_lineno(0, p.lineno(1))

    def p_sysargs_one(self, p):
        'sysargs : sysarg'
        p[0] = (p[1],)
        p.set_lineno(0, p.lineno(1))

    def p_sysargs_empty(self, p):
        'sysargs : empty'
        p[0] = ()

    def p_sysarg(self, p):
        'sysarg : expression'
        p[0] = p[1]
```

```
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_function(self, p):
    'function : FUNCTION width ID SEMICOLON function_statement ENDFUNCTION'
    p[0] = Function(p[3], p[2], p[5], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_function_nowidth(self, p):
    'function : FUNCTION ID SEMICOLON function_statement ENDFUNCTION'
    p[0] = Function(p[2],
                    Width(IntConst('0', lineno=p.lineno(1)),
                          IntConst('0', lineno=p.lineno(1)),
                          lineno=p.lineno(1)),
                    p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_function_statement(self, p):
    'function_statement : funcvardecls function_calc'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_funcvardecls(self, p):
    'funcvardecls : funcvardecls funcvardecl'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_funcvardecls_one(self, p):
    'funcvardecls : funcvardecl'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_funcvardecl(self, p):
    """funcvardecl : decl
    | integerdecl
    """
    if isinstance(p[1], Decl):
        for r in p[1].list:
            if (not isinstance(r, Input) and not isinstance(r, Reg)
                    and not isinstance(r, Integer)):
                raise ParseError("Syntax Error")
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_function_calc(self, p):
    """function_calc : blocking_substitution
    | if_statement
    | for_statement
    | while_statement
    | case_statement
    | casex_statement
    | block
    | namedblock
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_functioncall(self, p):
    'functioncall : identifier LPAREN func_args RPAREN'
    p[0] = FunctionCall(p[1], p[3], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_func_args(self, p):
    'func_args : func_args COMMA expression'
    p[0] = p[1] + (p[3],)
    p.set_lineno(0, p.lineno(1))

def p_func_args_one(self, p):
    'func_args : expression'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_func_args_empty(self, p):
    'func_args : empty'
    p[0] = ()

# ------------------------------------------------------------------------
def p_task(self, p):
```

```
    'task : TASK ID SEMICOLON task_statement ENDTASK'
    p[0] = Task(p[2], p[4], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_task_statement(self, p):
    'task_statement : taskvardecls task_calc'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_taskvardecls(self, p):
    'taskvardecls : taskvardecls taskvardecl'
    p[0] = p[1] + (p[2],)
    p.set_lineno(0, p.lineno(1))

def p_taskvardecls_one(self, p):
    'taskvardecls : taskvardecl'
    p[0] = (p[1],)
    p.set_lineno(0, p.lineno(1))

def p_taskvardecls_empty(self, p):
    'taskvardecls : empty'
    p[0] = ()

def p_taskvardecl(self, p):
    """taskvardecl : decl
    | integerdecl
    """
    if isinstance(p[1], Decl):
        for r in p[1].list:
            if (not isinstance(r, Input) and not isinstance(r, Reg)
                    and not isinstance(r, Integer)):
                raise ParseError("Syntax Error")
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

def p_task_calc(self, p):
    """task_calc : blocking_substitution
    | if_statement
    | for_statement
    | while_statement
    | case_statement
    | casex_statement
    | block
    | namedblock
    """
    p[0] = p[1]
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_identifier(self, p):
    'identifier : ID'
    p[0] = Identifier(p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_scope_identifier(self, p):
    'identifier : scope ID'
    p[0] = Identifier(p[2], p[1], lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_scope(self, p):
    'scope : identifier DOT'
    scope = () if p[1].scope is None else p[1].scope.labellist
    p[0] = IdentifierScope(
        scope + (IdentifierScopeLabel(p[1].name, lineno=p.lineno(1)),), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

def p_scope_pointer(self, p):
    'scope : pointer DOT'
    scope = () if p[1].var.scope is None else p[1].var.scope.labellist
    p[0] = IdentifierScope(scope + (IdentifierScopeLabel(p[1].var.name,
                                                         p[1].ptr, lineno=p.lineno(1)),), lineno=p.lineno(1))
    p.set_lineno(0, p.lineno(1))

# ------------------------------------------------------------------------
def p_disable(self, p):
    'disable : DISABLE ID'
    p[0] = Disable(p[2], lineno=p.lineno(1))
```

```
        p.set_lineno(0, p.lineno(1))

    # ------------------------------------------------------------------------
    def p_single_statement_delays(self, p):
        'single_statement : DELAY expression SEMICOLON'
        p[0] = SingleStatement(DelayStatement(
            p[2], lineno=p.lineno(1)), lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_single_statement_systemcall(self, p):
        'single_statement : systemcall SEMICOLON'
        p[0] = SingleStatement(p[1], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    def p_single_statement_disable(self, p):
        'single_statement : disable SEMICOLON'
        p[0] = SingleStatement(p[1], lineno=p.lineno(1))
        p.set_lineno(0, p.lineno(1))

    # fix me: to support task-call-statement
    # def p_single_statement_taskcall(self, p):
    #     'single_statement : functioncall SEMICOLON'
    #     p[0] = SingleStatement(p[1], lineno=p.lineno(1))
    #     p.set_lineno(0, p.lineno(1))

    # def p_single_statement_taskcall_empty(self, p):
    #     'single_statement : taskcall SEMICOLON'
    #     p[0] = SingleStatement(p[1], lineno=p.lineno(1))
    #     p.set_lineno(0, p.lineno(1))

    # def p_taskcall_empty(self, p):
    #     'taskcall : identifier'
    #     p[0] = FunctionCall(p[1], (), lineno=p.lineno(1))
    #     p.set_lineno(0, p.lineno(1))

    # ------------------------------------------------------------------------
    def p_empty(self, p):
        'empty : '
        pass

    # ------------------------------------------------------------------------
    def p_error(self, p):
        print("Syntax error")
        if p:
            self._parse_error(
                'before: %s' % p.value,
                self._coord(p.lineno))
        else:
            self._parse_error('At end of input', '')


class VerilogCodeParser(object):

    def __init__(self, filelist, preprocess_output='preprocess.output',
                 preprocess_include=None,
                 preprocess_define=None):
        self.preprocess_output = preprocess_output
        self.directives = ()
        self.preprocessor = VerilogPreprocessor(filelist, preprocess_output,
                                                preprocess_include,
                                                preprocess_define)
        self.parser = VerilogParser()

    def preprocess(self):
        self.preprocessor.preprocess()
        text = open(self.preprocess_output).read()
        os.remove(self.preprocess_output)
        return text

    def parse(self, preprocess_output='preprocess.output', debug=0):
        text = self.preprocess()
        ast = self.parser.parse(text, debug=debug)
        self.directives = self.parser.get_directives()
        return ast

    def get_directives(self):
        return self.directives
```

```
def parse(filelist, preprocess_include=None, preprocess_define=None):
    codeparser = VerilogCodeParser(filelist,
                                   preprocess_include=preprocess_include,
                                   preprocess_define=preprocess_define)
    ast = codeparser.parse()
    directives = codeparser.get_directives()
    return ast, directives
```

REFERENCES

[1] G. E. Moore *et al.*, *Cramming more components onto integrated circuits*, 1965.

[2] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., 2007, p. 400.

[3] N. R. Storey, *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[4] M. L. Shooman, "Reliability of computer systems and networks: Fault tolerance," *Analysis, and Design, Wiley-Interscience*, 2001.

[5] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[6] M. Nourani and A. Radhakrishnan, "Power-supply noise in socs: Atpg, estimation and control," in *IEEE International Conference on Test, 2005.*, IEEE, 2005, 10–pp.

[7] M. Elgamel and M. Bayoumi, "Noise analysis and design in deep submicron technology," in *The Electrical Engineering Handbook*, Academic Press, 2005, pp. 299–310.

[8] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[9] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[10] J. Schwank, M. Shaneyfelt, D. Fleetwood, J. Felix, P. Dodd, P. Paillet, and V. Ferlet-Cavrois, "Radiation effects in mos oxides," *Nuclear Science, IEEE Transactions on*, vol. 55, pp. 1833 –1853, Sep. 2008.

[11] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, ISBN: 9780080558325, 9780123695291.

[12] H. Fujiwara, *Logic Testing and Design for Testability*. MIT Press, 1985.

[13] N. R. Shnidman, W. H. Mangione-Smith, and M. Potkonjak, "On-line fault detection for bus-based field programmable gate arrays," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 4, pp. 656–666, 1998.

[14] E. Sentovich, *SIS: A System for Sequential Circuit Synthesis*. Electronics Research Laboratory, College of Engineering, University of California, 1992.

[15] A. Pal, "Cellular realization of tsc checkers for error detecting codes," in *Computer and Communication Systems, 1990. IEEE TENCON'90., 1990 IEEE Region 10 Conference on*, 1990, 687–691 vol.2.

[16] M. Nicolaidis, "Carry checking/parity prediction adders and alus," *Ieee Transactions on Very Large Scale Integration (Vlsi) Systems*, vol. 11, no. 1, pp. 121–128, 2003.

[17] M. Gssel, V. Ocheretny, E. Sogomonyan, and D. Marienfeld, *New Methods of Concurrent Checking*. Springer Netherlands, 2008.

[18] S. Almukhaizim, P. Drineas, and Y. Makris, "Entropy-driven parity-tree selection for low-overhead concurrent error detection in finite state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 8, pp. 1547–1554, 2006.

[19] S. Mitra and E. J. McCluskey, "Which concurrent error detection scheme to choose?" In *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*, 2000, pp. 985–994.

[20] S. Toutounchi and A. Lai, "Fpga test and coverage," in *Proceedings. International Test Conference*, 2002, pp. 599–607.

[21] J. O. Hamblen and M. D. Furman, *Rapid Prototyping of Digital Systems: A Tutorial Approach*. Kluwer Academic Publishers, 2001, p. 270.

[22] S.-B. Ko and J.-C. Lo, "Efficient realization of parity prediction functions in fpgas," *Journal of Electronic Testing*, vol. 20, no. 5, pp. 489–499, 2004.

[23] M. Tahoori, "Application-dependent testing of fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 1024–1033, 2006.

[24] M. Abramovici, C. Strond, C. Hamilton, S. Wijesuriya, and V. Verma, "Using roving stars for on-line testing and diagnosis of fpgas in fault-tolerant applications," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, 1999, pp. 973–982.

[25] S. Mitra, W.-J. Huang, N. R. Saxena, S.-Y. Yu, and E. J. McCluskey, "Reconfigurable architecture for autonomous self-repair," *IEEE Design & Test of Computers*, vol. 21, no. 3, pp. 228–240, 2004.

[26] I. Herrera-Alzu and M. Lopez-Vallejo, "Design techniques for xilinx virtex fpga configuration memory scrubbers," *IEEE transactions on Nuclear Science*, vol. 60, no. 1, pp. 376–385, 2013.

[27] A. Stoddard, A. Gruwell, P. Zabriskie, and M. J. Wirthlin, "A hybrid approach to fpga configuration scrubbing," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 497–503, 2017.

[28] M. Berg, C Poivey, D Petrick, D Espinosa, A. Lesea, K. LaBel, M Friendlich, H Kim, and A. Phan, "Effectiveness of internal versus external seu scrubbing mitigation strategies in a xilinx fpga: Design, test, and analysis," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, pp. 2259–2266, 2008.

[29] V. Dumitriu, L. Kirischian, and V. Kirischian, "Run-time recovery mechanism for transient and permanent hardware faults based on distributed, self-organized dynamic partially reconfigurable systems," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2835–2847, 2016.

[30] S. Kim, H. Chu, I. Yang, S. Hong, S. H. Jung, and K.-H. Cho, "A hierarchical self-repairing architecture for fast fault recovery of digital systems inspired from paralogous gene regulatory circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 12, pp. 2315–2328, 2012.

[31] R. Tessier and H. Giza, "Balancing logic utilization and area efficiency in fpgas," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, R. W. Hartenstein and H. Grünbacher, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 535–544.

[32] G. E. Moore, "Cramming more components onto integrated circuits (reprinted from electronics, pg 114-117, april 19, 1965)," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[33] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *Ibm Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[34] C. Carmichael, "Triple module redundancy design techniques for virtex fpgas," *Xilinx Application Note XAPP197*, vol. 1, 2001.

[35] J. Han, E. R. Boykin, H. Chen, J. H. Liang, and J. A. B. Fortes, "On the reliability of computational structures using majority logic," *IEEE Transactions on Nanotechnology*, vol. 10, no. 5, pp. 1099–1112, 2011.

[36] F. P. Mathur and A. Avizienis, "Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair," in *AFIPS Spring Joint Computing Conference*, 1970.

[37] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, C. Braun, M. E. Imhof, H. Wunderlich, and J. Henkel, "Module diversification: Fault tolerance and aging mitigation for runtime reconfigurable architectures," in *2013 IEEE International Test Conference (ITC)*, 2013, pp. 1–10.

[38] H. Baig and J. Lee, "An island-style-routing compatible fault-tolerant fpga architecture with self-repairing capabilities," in *2012 International Conference on Field-Programmable Technology*, 2012, pp. 301–304.

[39] R. F. DeMara, J. Lee, R. Al-Haddad, R. S. Oreifej, R. Ashraf, B. Stensrud, and M. Quist, *Dynamic Partial Reconfiguration Approach to the Design of Sustainable Edge Detectors*. 2010, pp. 49–58.

[40] R. S. Oreifej, C. A. Sharma, and R. F. DeMara, "Expediting ga-based evolution using group testing techniques for reconfigurable hardware," in *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, 2006, pp. 1–8.

[41] R. Giordano, D. Barbieri, S. Perrella, R. Catalano, and G. Milluzzo, "Configuration self-repair in xilinx fpgas," *IEEE Transactions on Nuclear Science*, 2018.

[42] D. C. Keezer and J. Yang, "Biologically inspired hierarchical structure for self-repairing fpgas," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, IEEE, 2017, pp. 1–8.

[43] Xilinx, "Device reliability report," Tech. Rep., 2016.

[44] H. Quinn, "Radiation effects in reconfigurable fpgas," *Semiconductor Science and Technology*, vol. 32, no. 4, 2017.

[45] A. Ceschia, A. Violante, M. S. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, "Identification and classification of single-event upsets in the configuration memory of sram-based fpgas (vol 50, pg 2088, 2003)," *IEEE Transactions on Nuclear Science*, vol. 51, no. 2, pp. 328–328, 2004.

[46] P. Adell and G. Allen, "Assessing and mitigating radiation effects in xilinx fpgas," Pasadena, CA: Jet Propulsion Laboratory, California Institute of Technology, 2008, Tech. Rep., 2008.

[47] Xilinx, "Vivado design suite user guide - partial reconfiguration," 2017.

[48] K. Vipin and S. A. Fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 72, 2018.

[49] Z. Seifoori, B. Khaleghi, and H. Asadi, "Introduction to emerging sram-based fpga architectures in dark silicon era," in. Jan. 2018.

[50] N. Mark and M. Peattie, "Using a microprocessor to configure xilinx fpgas via slave serial or selectmap mode," Jan. 2001.

[51] V. Lai and O. Diessel, "Icap-i: A reusable interface for the internal reconfiguration of xilinx fpgas," Jan. 2010, pp. 357 –360.

[52] Xilinx, "Xc6200 field programmable gate arrays product description," 1997.

[53] ——, "Xilinx ds031 virtex-ii platform fpgas: Complete data sheet," 2003.

[54] ——, "Ug070: Virtex-4 fpga user guide," 2008.

[55] ——, "Xapp151: Virtex series configuration architecture user guide," 2004.

[56] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," in *2006 International Conference on Field Programmable Logic and Applications*, IEEE, 2006, pp. 1–6.

[57] M. Fitzgerald, *Introducing Regular Expressions: Unraveling Regular Expressions, Step-by-Step*. O'Reilly Media, 2012.

[58] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd. New York, NY, USA: Cambridge University Press, 2003, ISBN: 052182060X.

[59] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[60] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[61] D. Beazley, "Ply (python lex-yacc)," *See http://www. dabeaz. com/ply*, 2001.

[62] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040, Springer International Publishing, 2015, pp. 451–460.

[63] F. Corno, M. S. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *IEEE Design Test of Computers*, vol. 17, no. 3, pp. 44–53, 2000.

[64] S. R. Welke, B. W. Johnson, and J. H. Aylor, "Reliability modeling of hardware/-software systems," *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 413–418, 1995.

[65] M. Rausand, "Reliability of safety-critical systems," *John Wiley&Sons*, 2014.

[66] D. P. Siewiorek and R. S. Swarz, *Reliable computer systems: design and evaluation*. AK Peters/CRC Press, 1998.

[67] S. Scharoba, M. Schölzel, T. Koal, and H. T. Vierhaus, "On reliability estimation for combined transient and permanent fault handling," in *2014 14th Biennial Baltic Electronic Conference (BEC)*, IEEE, 2014, pp. 73–76.

[68] I. Koren and S. Y. H. Su, "Reliability analysis of n-modular redundancy systems with intermittent and permanent faults," *IEEE Transactions on Computers*, no. 7, pp. 514–520, 1979.

[69] D. McMurtrey, K. S. Morgan, B. Pratt, and M. J. Wirthlin, "Estimating tmr reliability on fpgas using markov models," 2008.

[70] K. S. Trivedi, *Probability & Statistics with Reliability, Queuing and Computer Science Applications*. PHI Learning Pvt. Limited, 2011.

[71] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 5–14.

[72] S. A. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," *2008 Symposium on Application Specific Processors*, pp. 101–+, 2008.

[73] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[74] G. Marsaglia *et al.*, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

VITA

Jingchi Yang was born in 1990 in China. He joined a dual degree program and received one B.S. degree in electrical engineering from the Hohai University in China and one B.S. degree from the University of Lille 1 in France in 2012. He continued on to receive the M.S. degree in microelectronics and nanotechnology from the University of Lille 1 in 2014, and the M.S. degree in electrical and computer engineering from Georgia Institute of Technology in 2015.

In the Spring of 2015, Dr. Yang joined the High-Speed Digital Test Lab at Georgia Tech under the guidance of Dr. David Keezer in order to purse the Ph.D. degree in electrical and computer engineering. His primary areas of expertise are fault-tolerant and self-repair digital systems design. He has over five years of experience in register-transfer level (RTL) design and FPGA development. In the Fall of 2019, Dr. Yang received the Ph.D. degree in electrical and computer engineering from Georgia Tech.